An Automatic Microprogramming System

by

Wu Kam-Wah

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of Master of Philosophy in Electronics

The Chinese University of Hong Kong

March 1985

Acknowledgement

I would like to express my deepest gratitude towards Prof. T. C. Chen for his stimulating discussion on the subject. Particular thanks are also dedicated to Mr. H. L. Lin and Dr. T. S. Yum for their giving advices on the preparation of this thesis. Last, but not the least, I would like to thank Dr. G. D. Chandler, Principal Lecturer of Computer Studies Department of the HK Polytechnic, for his many valuable suggestions and comments, lest this thesis could not be duly completed.

CONTENT

Abstract

Generating microcode from higher level microprograms has long been a tough issue for it involves various conflicting factors like economic considerations, ease on user microprogramming aspect and the generality of the microcode generation system, etc.

The aim of introducing the AMPS (automatic microprogramming system) is to present an approach towards automatic microcode generation. Under such an approach, rather than constructing a translator for the microprogramming language, only an once-off description of the microarchitecture is required. Bases on this description the AMPS then generates the necessary functions to translate source higher level microprograms to microcode. In fact, the above process of microcode generation also signifies the meaning of "automatic".

To realize the proposed approach, two languages are developed, namely AMPL (a microprogramming language) and MIMOL (a microinstruction modelling language). The design of AMPL is basically influenced by Dasgupta's S* [1] but augmented with some additional important language constructs like macros, non-return call of function routine commands, etc. It is applicable to different microarchitectures by adopting Dasgupta's method of instantiation [1]. MIMOL, on the other hand, is designed to allow users to define a microarchitecture hierarchically through the declarations of microinstruction intra- and inter-field information. MIMOL also provides a formalism to describe various

microarchitectures in a completely machine independent manner.

Certain modest yet fundamental goals are achieved by the AMPS designed, namely,

(i) Retargetability, i.e. the applicability to different machines having similar microarchitecture to the Chen's machine [2];

(ii) User Microprogrammability, i.e. users can practise higher level microprogramming without much difficulties;

(iii) Transportability, i.e. the AMPS can be implemented on different machines without much difficulties;

(iv) Integrity in the development of microcode. This is achieved as both AMPL and MIMOL are applied in a coherent fashion for microcode development in AMPS;

(v) Automatic generation of microcode, i.e. no dedicated translator for the microprogramming language of any particular machine is required to construct for the generation of microcode.

To demonstrate the feasibility of the AMPS, a prototype system developed for the Chen's machine [2] is implemented using VS/APL. Microcode is also generated to emulate an architecture similar to that of the IBM S/360. VS/APL, contrasting to most other single-user environment for microcode development, provides a mechanism whereby a class of students can learn microprogramming and microarchitecture in an efficient manner. Thus, together with the hardware version of the Chen's machine [3], the current implemented AMPS also carries a high pedagogic value.

Chapter 1. Introduction

1.1 Motivation

Although microprogramming was introduced by Wilkes in 1951 [4], the first extensive commercial microprogramming project was the design and implementation of the IBM S/360 series of compatible processors in the early sixties [5]. Since then microprogramming has grown rapidly and is now a widely accepted processor implementation technique.

With this growth in microprogramming comes the requirement for better microprogramming production method. Just as the need for software to be produced more quickly, cheaply and reliably has been recognized, the need for these characteristics in microprogram development is obvious. There are at least three major reasons for this need. The first is that many microprograms such as those used in the implementation of an instruction set are still being committed to read only storage (ROS) or read only memory (ROM). Errors in these applications are expensive to repair if a proper set of tools is not available. Second, the increased migration of programs from other programming levels into firmware thus leads to larger microprograms. An example is the support of high level languages by the application of microprogramming--- both in terms of processing and execution of these languages. Some applications are actually reported such as the implementation of an APL machine by Hassitt [6] and the implementation of EULER by Weber [7]. These [6,7] all indicate the fact that it is promising to execute high level language

via microprogramming provided that the microprogrammed high level language primitives are properly chosen. Another example is the microprogramming support of the operating systems as reported in the [8,9]. As a result, more microprogramming activities are anticipated. The third reason is the growth of user microprogramming brought about by the availability of processors with writable control store (WCS) or the alterable control memory (ACM) and the introduction of what so called the "bit-slice" products.

Therefore the requirement for an efficient microcode production system is imminent. Accordingly, a system which can generate microcode automatically from higher level microprograms will greatly facilitate user microprogramming. Motivated by this, the research project for the design and implementation of such an automatic microprogramming system is so initiated.

1.2 Definition of Terms

Many terms used in this thesis may mean differently to different people. Therefore, for the sake of consistency and clarity, we present our definitions for these terms as follows:

a) Operation unit-- this is the data manipulation part of a digital computer.

b) Control unit-- this is the part of the digital computer exercises control on the operation unit. In other words, it is responsible for the initiation and sequencing of the primitive operations of the operation unit.

c) Microarchitecture-- this term refers to the structure of a microprogrammable machine that can be accessed and manipulated by the microprogrammer. In other words we restrict our use of the word "microarchitecture" to describe:

i) the set of microinstructions through which the machine can be controlled and manipulated by the microprogrammer, and

ii) the machine organization that is visible to the microprogrammer.

d) Micro-operation-- this term refers to the elemental operation that is initiated by a field of the microinstruction. Micro-operation (herein after called MOP), therefore, is different from the primitive operations of the operation unit. MOP is effectively the most primitive action available to the microprogrammer.

e) Host machine-- this is the microprogrammed machine where the final microcode produced is to be resided and executed.

g) Base machine-- this is the machine where the AMPS is placed and microcode for the host machine is developed.

Chapter 2. Introducing AMPS

2.1 Design Goals

Generally speaking, due to the wide spectrum of microarchitectures (from simple monophase to complex polyphase) [10], designing an AMPS is a highly complex task and it involves the study of

a) microarchitecture design;

b) language modelling for microprogramming;

c) machine description methodology;

d) translator technology;

e) microprogram optimization techniques; and

f) microprogramming environment in which the AMPS embeds for the development and production of microcode.

Therefore, rather than being too ambitious to design an "all-perfect" AMPS, that requires an extensive study of the above topics, the scope of this research is to design an AMPS which attains the following modest yet fundamental goals:

(i) Retargetability, i.e. the AMPS should be a general system applicable to different machines within the same class of microarchitecture. This means that the designed AMPS should be capable of "cross-production" of microcode for different host machines.

(ii) User Microprogrammability, i.e. the AMPS should allow users to have higher level microprogramming rather than involved in the

intricate idiosyncracies of the machine microprogrammed. This, in particular refers to freeing the users from knowing the particulars of the control store organization. The microprogramming language used should also be easy to understand and use.

(iii) Transportability, i.e. the AMPS should pose little problem to its implemention. In other words the AMPS should be machine independent and can be implemented in different base machines for the development of microcode.

(iv) Integrity in the development of microcode, i.e. different constituents of the AMPS should be functioning in a coherent fashion for the development of microcode. For this can aliminate the requirement of "multi-environments" for the development of microcode.

(v) "Automatic-generation" of microcode. "Automatic" here implies

1) Higher level microprogramming at the source level without the need of users' knowledge about the control store organization. This basically coincides with a part of point (ii) above.

2) There is no need to construct any language translator for the host machine. In turn, users only need to supply a description of the host microarchitecture to the AMPS. The AMPS should then generate the necessary language translator and takes the care of translating the source higher level microprograms to microcode of the host

machine concerned.

Consequently, the fore-mentioned a), b), c), d), and f) are the main issues concerned in designing our AMPS.

2.2 AMPS Organization

Conceptually, the basic organization of an AMPS is as shown in Fig.2.1.

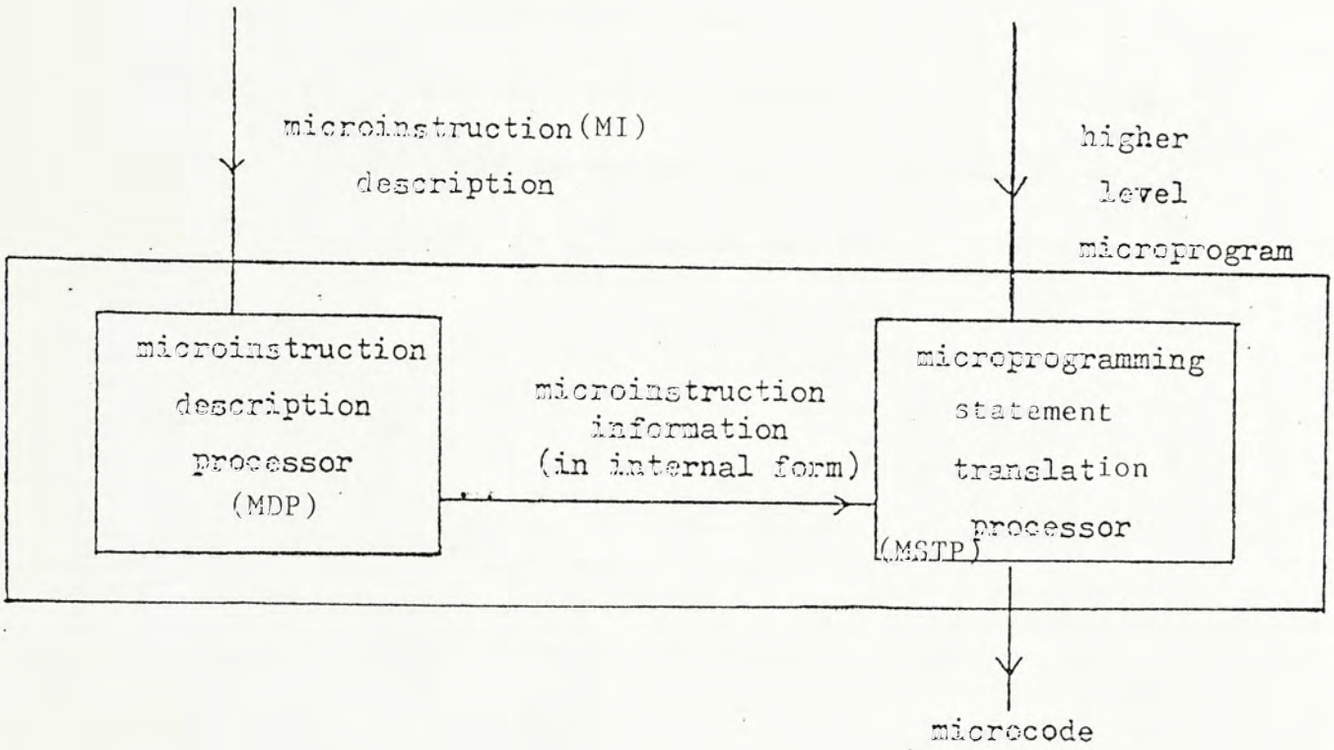


Fig.2.1 AMPS basic organization

An AMPS accepts two primary inputs: the source microprogram written in a higher level microprogramming language, and the description of the microprogrammable machine microinstruction.

The microinstruction description processor is the part of the AMPS that accepts the input microinstruction description and transforms it into the proper internal representation. This internal representation will then be used by the translation processor for the generation of microcode.

It should be noted that an AMPS is only functionally divided into these two major components. However, how the AMPS is actually constructed depends on how one approaches to achieve automatic microcode generation. Yet, to the best knowledge of the author there is no consensus method nor approach proposed towards such an aim. Nevertheless, in the coming section such an attempt is presented.

2.3 An Approach Towards Automatic Generation of Microcode

The approach proposed to achieve automatic microcode generation is based on the following two observations.

(a) Many fields of a microinstruction, in fact, are inter-related. The way how a microinstruction addresses or initiates microoperations, is, as a matter of fact, solely achieved by the setting up of certain bit patterns in the MI fields (this is also known as microcoding). And due to the particular nature of microoperations-- direct operations on machine hardware, MOPs are basically of the function/operation (including the branching operation) or the register-transfer types of operation. These machine basic operations, in turn, are very primitive and they usually simply involve the selection of operations/operands or the selection of destination/source registers or devices. As MOPs can usually be directly mapped with the fields of the MI, fields of the MI are thus implicitly related by the nature of the MOPs they are controlling or addressing. Suppose there are two fields responsible for the initiation of a register-transfer operation. These two fields may then be probably related by the source-destination relationship implied by the MOP they are triggering; as one field may be responsible for the source register select while the other acts as the destination register select field. Similarly, an operation-operand relation can also exist between two fields responsible for the initiation of a function/operation type of MOP (assumed mono-operand). With one field being responsible as an operation select field, the other field then is

responsible for the selection of operand for the operation selected. In cases where the MOP does not need any operand, it can still be treated as a function/operation type of MOP (for it simply is a null-operand microoperation). Then we may suppose the function or operation select field is related to a null field.

(b) The final translated microcode (or called the object code) can be treated as a piece of matrix with its elements bearing the value either 1 or 0.

Base on these two observations, we may then derive some suppositions. These suppositions also serve as the premise for the approach proposed. These suppositions are:

1) Under observation (a), it implies that microinstruction fields, with their implicitly existing operation-operand or destination-source relation, could be depicted syntactically as general programming languages.
2) In other words, rather than as conventionally describing microinstruction fields in an unrelated, field by field basis; a microinstruction description language can be so designed that microinstruction fields relations are described in an abstract manner resembling general programming languages. Hopefully, if the inter-field description could appear and be applied in a coherent fashion with the microprogramming language used, the microcode generation pr ss v .. then be facilitated.

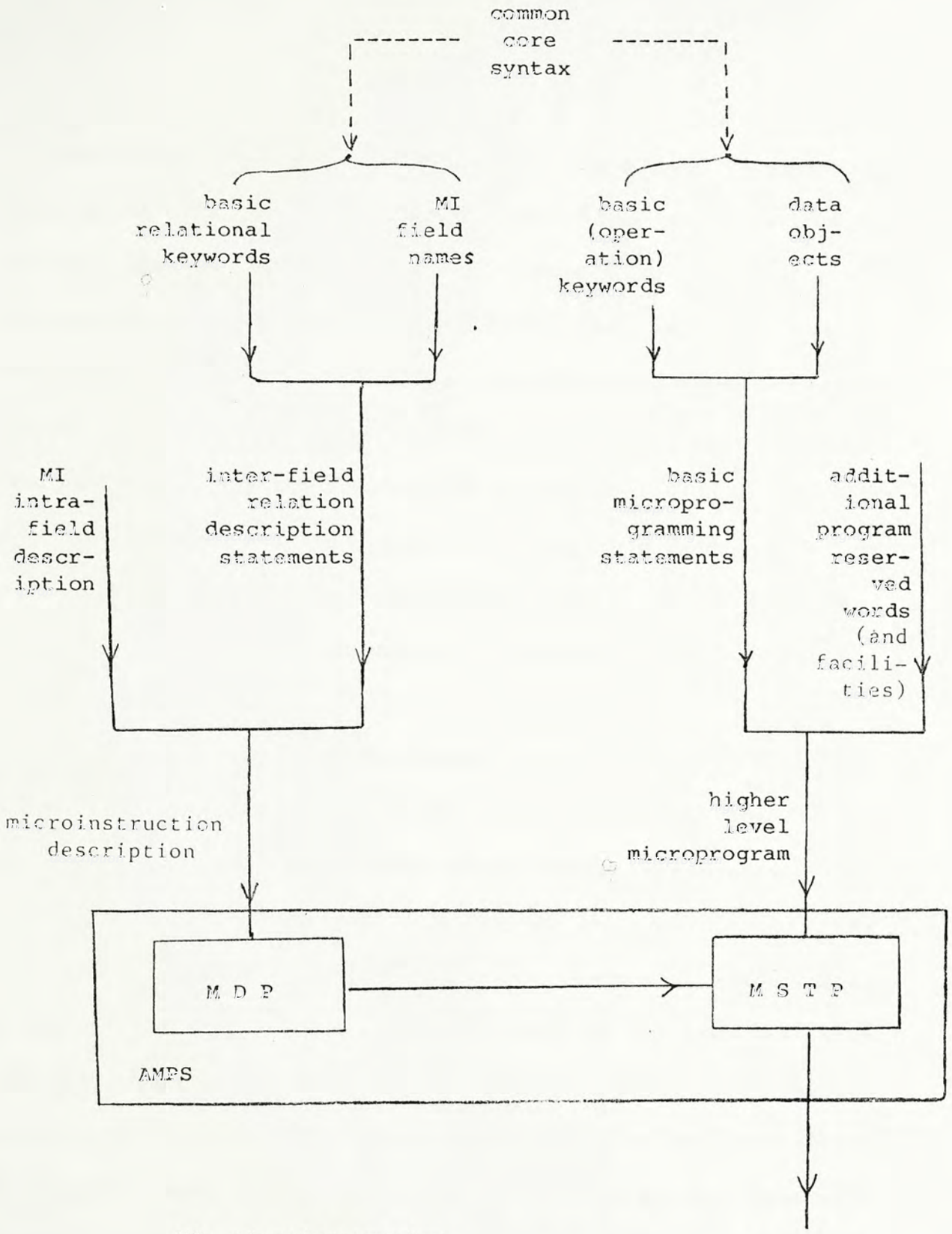
(This point will become clear in latter paragraphs).

3) Under observation (b), the process of generation of microcode is given another dimension. Rather than conventionally treating the final microcode as a result of the process of pattern matching and recognition; the microcode can also be generated as a result of some bit-setting functions on a presumed object code matrix.

4) That is, rather than using conventional method of parsing and syntax analysing, the language translator, on accepting the source programming statements, initiates (if we deliberately design it so) some functions with their execution effects a setting of values to a presumed final object code matrix piece, the generation of microcode would then be very efficient. (This process of microcode generation will be discussed in full with example later on).

Having all the preliminaries stated, we now go into the details of the approach proposed towards automatic generation of microcode.

The basic concept structure of the approach is described in Fig.2.2.



MI: microinstruction
MDP : microinstruction description processor
MSTP: microprogramming statement translation processor

Fig.2.2 The Basic Concept Structure for the Approach Towards Automatic Microcode Generation

Referring to Fig.2.2, the microinstruction intra-field description is the description on each of the microinstruction fields details including their names (or identifiers), corresponding positions in the MI word, and relevant machine resources as referred by them, etc. The basic relational keywords for MI inter-field relations description are the type of keywords that can depict the basic relations between MI fields as accorded to the basic machine operations they are addressing or controlling. These basic keywords also have a close resemblance to those basic (operation) keywords employed for microprogramming.

As shown in Fig.2.2, the approach employs a core syntax for the MI inter-field description and microprogramming. That is, a discipline is posed (as accorded to the common core syntax) such that the MI inter-field description statements and the basic microprogramming statements are resembling each other syntactically. (The core syntax employed for our AMPS is also clearly shown in sec.3.2.4 and sec.4.2.4). There are certain advantages in employing a common core syntax for MI inter-field description and microprogramming. First, it brings coherence between the MI inter-field relation statements and source microprogramming statements. A correlation between user microprogramming and microinstruction description can also be established. Moreover, under the common the syntax, a unified picture between MI description and microprogram description (as microprogramming statements, in fact, are describing the

microprogram) can thus be obtained. In addition, the adoption of a common core syntax also brings convenience to the construction of the MSTP (microprogramming statement translation processor) as constructs of the source microprogramming statements can directly be referenced with their counterparts in the MI description under the common syntax.

During translation, the source microprogramming statements data objects are identified. (These data objects, as will be illustrated in later chapters, actually corresponds to the host machine resources declared in the intra-field description). At the mean time the keywords (including the program reserved words and necessary operation keywords, etc.) are also sorted out from the source statements. In fact, the microprogramming statement translation processor (see Fig.2.1), on receiving these operation keywords, directly executes the relevant bit-setting functions corresponding to them, and effects in the setting of bits on the presumed final object code matrix. If the bit-setting functions in the translation processor are designed in such a fashion that their names correspond to the "would-be-accepted" keywords, the process of microcode generation will be highly facilitated. For the job of recognizing these keywords has now become the job of recognizing the function names exist in the system/environment the AMPS is embedding. As a result it becomes the background system task and is passed to the background system/environment rather than the AMPS! Therefore from the AMPS viewpoint, the generation of microcode can be very efficient. In reality, this very method is also exploite the pproach proposed for

automatic microcode generation. We also call such a way of translation by the name "translation by direct execution". This concept of translation is further illustrated in Fig.2.3 below.

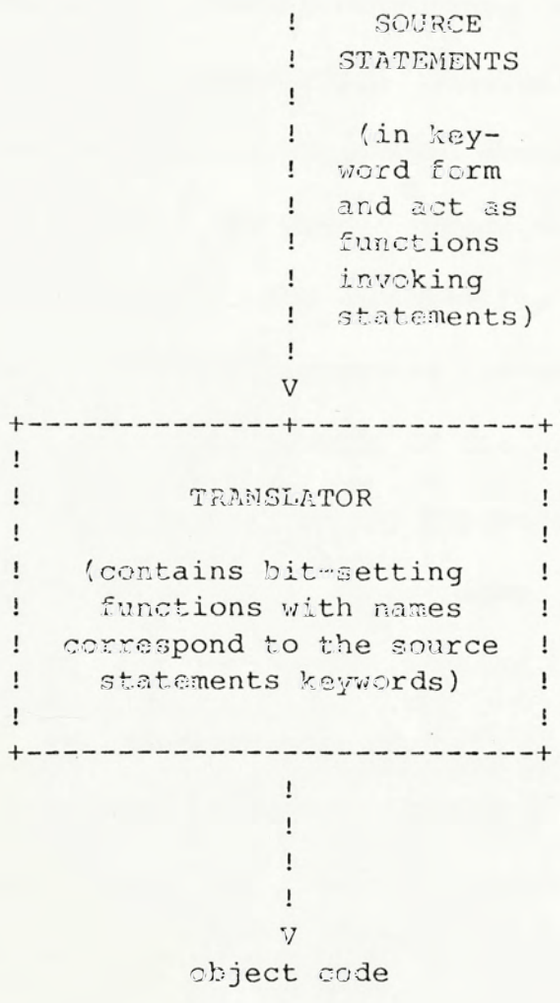


Fig.2.3 The Concept of Translation by Execution

Nevertheless, in order to have "real" automatic generation of microde, one more step is required. That is, we should have the bit-setting functions contained in the microprogramming statement translation processor (see Fig.2.1) be generated within the AMPS. In other words, rather than writing the bit-setting functions one by one for the translation processor, the translation processor, after receiving the microinstruction information in an appropriate internal form from the microinstruction description processor, generates the bit-setting functions corresponding to these operation keywords. And on accepting these keywords the translation processor directly executes these keyword-corresponding bit-setting functions and effects in the setting of the bits of the presumed final object code matrix. Thus, the translation processor, rather than doing the job of parsing and analysing, does the job of proper functions generation and execution instead to achieve microcode generation. The microcode generation process is automatic for the bit-setting functions are created internal to and by the AMPS itself. Retargetability can also be achieved as microprogramming for different host machines only requires one to supply the corresponding machine microinstruction descriptions. (This point will further be elaborated in later sections).

One final note, this approach only provides a fundamental framework or guideline towards automatic microcode generation. And how it actually works may have deviations from one installation to another. Nev[illegible] hel[illegible] the basic principle,

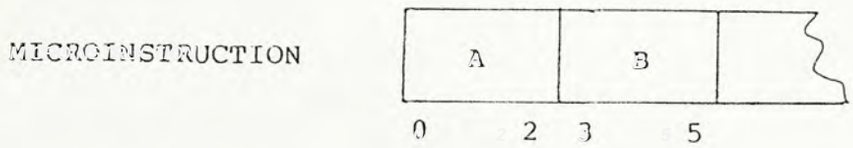
remains the same.

Below also shows in a step by step manner an example of how to apply the approach to achieve automatic microcode generation for a part of an hypothetic microinstruction in VS/APL.

EXAMPLE:

1) Assumption:

There are two fields named A and B in the MI with their positions in the MI as illustrated below



with field A at the MI bit positions 0, 1, 2 (or extending from bit position 0 to 2) and field B is at MI positions 3, 4, 5. Suppose A is the source and B is the destination register select fields with their corresponding relevant (selectable) registers (which are also noted as parts of the host machine resources) as follows:

	register	corresponding binary value in the field
field A :	R1	000
	R2	001
field B :	R3	010
	R4	011

2) Intra-field Information:

The intra-field information can be stored up in

VS/APL by the following VS/APL statements

```
R1<---0,0,0
R2<---0,0,1
R3<---0,1,0
R4<---0,1,1
A<---'MI[0,1,2]'
B<---'MI[3,4,5]'
```

where MI is an abbreviation for the microinstruction word.

3) Inter-field Information:

Assume the keyword "TO" is used to depict the register transfer relation between fields A and B then we may have

```
A TO B
```

as the inter-field relation description between A and B. The field names A and B in the above description statement thus are abstract representations of their corresponding selectable resources for the register-transfer relation existing among them. To keep with the basic objective of the proposed approach, we should have the following parallelism between source microprogramming basic statements and the MI inter-field description statement.

MI inter-field description statement		source microprogramming statement
A TO B	<----->	{R1 / R2} TO {R3 / R4}

In VS/APL the keyword "TO" used for inter-field description is actually a VS/APL function and may be written as follows:

```
     ∇ X TO Y
[1]   TO<--15↑'X TO Y'
[2]   TO<--TO,[0.5] 15↑X,'<--X'
[3]   TO<--TO,[1] 15↑Y,'<--Y'
     ∇
```

During the execution of the function "TO" during MI interfield description, the 1st function statement creates a 15-character row named TO with content:

```
X  TO  Y
```

The 2nd function statement expand TO to a matrix

```
X  TO  Y
MI[0,1,2]<--X
```

The 3rd function statement further expand TO to

```
X  TO  Y
MI[0,1,2]<--X
MI[3,4,5]<--Y
```

In fact TO has now become a VS/APL expression (in a character matrix form) for a latent MI word bit-setting

function.

4) Automatic Translation by Direct Execution

During the actual translation, we can invoke a VS/APL function (i.e. a program) containing the statement

⎕EX 'TO' (can be executed in both system and/or VS/APL function levels)

to initialize the translation environment. (The original function "TO" for inter-field description can easily be restored back in the current VS/APL workspace using)PCOPY or)COPY commands from the appropriate workspace storing the function). After initialization, we can then establish the bit-setting function bearing the same name as the keyword "TO" in the inter-field relation description statement. The establishing of the bit-setting function in VS/APL can easily be achieved by using the system function ⎕FX statement as follows:

⎕FX TO (can be executed in a VS/APL program)

Now, suppose we have a source microprogramming basic statement

R1 TO R3

inputting to the translation processor (see Fig.2.1). As already illustrated, this source statement will simply initiate the execution of the function named "TO" with R1, R3 substituting for the operands X and Y respectively. In essence, we have the following VS/APL statements

```
MI[0,1,2]<--0,0,0
MI[3,4,5]<--0,1,0
```

executed. If the variable MI above is actually a row of the presumed object code matrix, automatic generation of microcode is thus achieved.

Although the number of linking variables or flags for the installation programs may increase as the complexity of the microarchitecture, the fundamental concept behind however, remains. It should also be noted that although the example is illustrated in VS/APL, it does not necessarily imply that the concept is dependent and works only at the VS/APL-based environment.

In the following chapters, steps towards the realization of the proposed approach for automatic microcode generation through the design and construction of an AMPS are presented and discussed. These steps basically, are:

a) The design or choice of an higher level microprogramming language;

b) The design or choice of a microinstruction description language;

c) The design or choice of an environment in which the two fore-mentioned major components of an AMPS embeds (see Fig.2.1).

We now go into these steps one by one in details.

Chapter 3. Design of AMPL

3.1 Design Considerations

The first step to realize the proposed approach for automatic microcode generation is to design or choose a proper language for higher level microprogramming. As for this step, due to the uniqueness of the approach, we have developed the language ourselves, namely called AMPL (A Microprogramming Language). It is basically a Dasgupta's S*[1] influenced language. In fact, some parts of AMPL are actually adopted from S*. However, AMPL does differ to S*, e.g. AMPL uses mnemonics rather than operator symbols, AMPL also possesses (while S* lacks) some language facilities which are deemed important in "real" microprogramming context like the macro facilities, etc. (see later sections).

The main aspects concerning the design of AMPL are also discussed as below.

(a) The Resource Binding Problem

The first problem addressed by microprogramming is the resource binding problem. This problem can best be described by the following example.

Consider a BASIC program to perform matrix multiplication. The semantics of BASIC requires that code must be written to do this operation component by component. Therefore the object code produced for this program will have to take many (basic)

instructions to perform the task. Now consider a host machine having matrix multiplication hardware available, which, in fact, can also be utilized via proper (machine) instruction. Hence, instead of writing the previous BASIC program, one such specialized instruction would be adequate. To compare, the resulting BASIC program is surely inefficient. And with regard to this, no matter how the resulting translated microcode of the BASIC program is optimized, this inefficiency would not be eliminated.

One possible way to solve the above problem would be to construct a translator that would recognize that a segment of the source program was doing matrix multiplication. During translation, this source segment will then be recognized and converted into the specialized matrix-multiply (machine) instruction. But, there are various many ways of writing programs by different programmers. This approach would then imply that the translator have to memorize every possible such segments that would likely be doing the task of matrix multiplication. Of course, this is not feasible. Another solution would be to incorporate a matrix-multiply operator into the source language. This does solve the problem, but the resulting language is no longer the general machine independent HLL (High Level Language) BASIC! Moreover, this is not a general solution to the problem. The reason being that a language based on this principle would have to include every possible host resource of every possible host machine. This is obviously impossible!

The problem of how to recognize the similarities between source constructs and host machine resources and how to bind the source constructs to the appropriate host machine resources is called the resource binding problem.

The resource binding problem simply implies that contemporary HLLs cannot directly be used to substitute as microprogramming languages. For instance, the HLL FORTRAN does not have the constructs to allow users to access the stack of a minicomputer. Most conventional HLLs also do not have the constructs for describing microparallelisms, which however is a basic feature of microprograms.

The way how the resource binding problem is tackled by AMPL will become clear in latter discussion.

(b) Machine Specificity and Microarchitecture Variability

The second problem we encounter in developing AMPL is machine specificity. This problem basically follows the resource binding problem. While the nature of a program is to work on an abstract machine, a microprogram however, works on a specific piece of hardware. Thus it is operationally insignificant to talk of a microprogram running on an abstract machine. Say, while it is perfectly sensible to refer a program as implementing some particular interface (or function), it is operation-wise meaningless to talk of an emulator (i.e. a microprogram) without referring to the host machine on which the emulator runs. Therefore, the machine specific nature of a microprogram makes it contradictory to state microprogramming in a machine independent

manner (which however is our ideal) as conventional high level programming languages.

This problem is further complicated by the fact that available microarchitectures come with a profusion of shapes and sizes. In fact, it is really very difficult for us to express microprograms without any reference to the host machine involved. Thus, if a microprogramming language has to be machine-independent, it then have to be sufficiently general to cope with the variations in microarchitectures. And no matter how hard one strive to abstract and generalize the many components of different microarchitectures, certain different and peculiar components regarding to individual microarchitecture still remain. In our design of AMPL, this notion of microprogramming is also acknowledged and an approach, namely instantiation [1], is adopted for the constitution of a microprogramming language from AMPL for a particular machine M (see section 3.2).

The issues on machine specificity and microarchitecture variability are also extensively discussed in [1].

(c) Degree of Abstraction

In our design of AMPL, we have the following remarks concerning the two main aspects of abstraction, namely the data and the function aspects.

i) Data Abstraction-- A natural way to model a data resource is with a data type. In the microprogramming domain, however, the

extent to abstract data objects is very limited. For as previously mentioned, microprograms implement virtual machines on physical hosts. In other words, to whatever level the data objects of the microprogram is abstracted, a conformation with the host machine resources still remains. Nevertheless, under such a limitation, AMPL data type can still be chosen in a fashion that a "conveniently" high level of abstraction retains so that general programming methodologies can still be applied. Furthermore, these data types may also be allowed to model a wide range of data resources of different machines. Thus, a certain degree of machine independent characteristics may be achieved, although semantically each declared data object of a particular data type of the microprogram is machine dependent.

ii) Function Abstraction-- In order to model the functional elements of different host machines, AMPL should possess the ability for function abstraction. As will be shown, with the allowance of macro declaration and instantiation of machine pecularities, AMPL thus acquires both flexibility and extensibility in modelling a wide range of different functional elements of different machines.

(d) Machine Independence and Higher-Levelness

In designing AMPL, we have to be clear about the issues of machine independence and "higher-levelness". First, when we are speaking of machine independence, we are referring to host

machine independence, i.e. not the base machine by which the object microprogram is generated (see chapter 1). Second, when we are talking of host machine independence, we are not comparing with the machine independence of conventional high level languages. As we have pointed out, microprogramming involves the programming of the intimate hardware features of a microprogrammable machine. Thus, the user has to possess the knowledge of the host machine microarchitecture to a certain extent. Therefore, designing a completely machine independent microprogramming language is nearly impossible. However, we would like, here, to say that a language is well worth to be described as "machine independent" if it possesses certain machine independent characteristics in its syntactic and semantic structures which make it (not the microprogram) applicable to a wide range of microarchitectures. In this sense, "machine independence" is also important to microprogram modification. (For example, when we are writing emulators for the same target machine under different hosts of similar microarchitecture, previously written microprograms will then be useful).

About the interpretation of "higher-levelness", there are quite different interpretations about this term from different authors, for example [1] and [11]. In designing AMPL, we feel content that it is a higher level microprogramming language if it enables one to write structured and readable microprograms. And in our writing of AMPL microprograms, we assume, the microprogrammer has the machine organization chart

and microarchitecture information in hand when he writes his AMPL microprograms.

(e) User Microprogrammability

With the advance of hardware technology, more and more sophiscated hardware systems are produced. Microprogrammable processors (especially the horizontal microcoded ones) which allow users' involvement are also becoming more attractive for many special high speed applications, e.g. signal processing, etc. Such involvement of the users' part in microprogramming is also enhanced by the introduction of what so called the 'bit-slice' products. Thus, user microprogrammability has become one of the most important factors to be considered in designing AMPL. In fact, AMPL will be more user microprogrammable if it is easy to understand and use. As will be described, AMPL is basically designed as a mnemonic type language which can easily be understood via its keywords. At the same time, AMPL is not "symbol-dependent" of any particular language to facilitate its portability.

3.2 The Language AMPL

3.2.1 An Overview

The language AMPL is basically designed with an approach adopted from Dasgupta's schema concept [1]. Therefore the AMPL structure also has a resemblance to Dasgupta's S* [1] but with some important language constructs augmented, for example the macro and the non-return call of procedure commands, etc. The language AMPL thus, using Dasgupta's words, "denotes a framework for the development of microprogramming languages for machines of different microarchitectures". That is, rather than directly microprogram with the lanugage AMPL, a specific microprogramming language developed from AMPL for a specific machine is used. For example, given a machine M, a particular language AMPL(M) is obtained by extracting the specifications of AMPL and at the same time filling into it the specific particulars of M. Using Dasgupta's term, AMPL is "instantiated" into a particular language AMPL(M) on the basis of (or with respect to) M. In fact, instantiated versions of AMPL only differ in their elementary statements and implications between different microoperations (see section 4.2.6).

Essentially, AMPL is a mnemonic type language and can easily be grasped by its keywords. In addition, AMPL has also retained the flavours of conventional high level languages such as structured control constructs, so as to facilitate the writing of structured and readable microprograms. Besides, AMPL also incorporates constructs for describing micro-parallelism. Similar

to Dasgupta's S* [1], AMPL basically consists of

(i) a set of elementary constructs whose syntax and semantics are only partially defined;

(ii) a set of composite constructs by means of which control structures in AMPL programs are developed.

We now present the language schema AMPL in details.

3.2.2. Data Types and Data Objects

a) Data Types:

As AMPL is designed with the intent of being used for more than a single microarchitecture, the data types of AMPL are thus chosen such that they are general enough and applicable to different microarchitectures.

The data type in AMPL basically resembles those of the S* [1] , namely the primitive and the structured data types. The primitive data type in AMPL is the BIT which is assumed to hold only the value 0 or 1. While for the structured type, as its name suggests, is structured from this primitive. Given the data objects having the type BIT the arithmetic and Boolean operations depicted in section 3.3.3 can be applied to them.

Structured data types are basically obtained by structuring the BIT data type. And there are four such categories (keywords are underscored for clarity) :

1) The sequence: SEQ n (III.1)

where n is the integer denoting the number of bits in the sequence. A sequence of this form can then be accessed by indexing the sequence name with either an integer constant i or the name of some other data objects. This data type "sequence" can actually be used to model machine registers.

2) The matrix: MATRIX m,n (III.2)

where m,n are integers denoting the number of rows and columns of the matrix respectively. A row of a matrix of this

form can be accessed by indexing the matrix name with either an integer constant i or the name of some other data objects following by ";". This data type "matrix" is actually used to model machine register files or memory. However many machines generally allow only certain machine resources as the addressing element (called pointer usually) for its memory elements. Such explicit binding of matrix and its corresponding addressing elements can be denoted by the "POINTER <identifier>" in the below statement:

MATRIX m,n POINTER '<identifier>{,<identifier>}' (III.3)

with the <identifier> as the pointer. Note that the notation "{x}" indicate a list of zero or more elements of the type "X". This notation also works for latter expressions.

3) The stack:

STACK n OF <type> POINTER '<identifier>{,<identifier>}'

(III.4)

The stack data type here has its usual meaning to represent the stack of a machine. While n here refers to the depth of the stack, the <identifier> after the word POINTER refers to the element addressing the current top of the stack. The standard primitive operations {push, pop} also go along with the stack data type. The <type> above may refer to the primitive type BIT or structured type SEQ (III.1).

4) The tuple: TUPLE '<identifier 1>' OF <type 1>

.

.

.

'<identifier k>' OF <type k>

ENDTUP (III.5)

The tuple here is basically adopted from S* [1]. Besides, it looks very like the Pascal record and is consisting of a number of fields of <type 1>,..., <type k>, repectively. Any valid operations on an individual field of the tuple are determined by its corresponding type and the nature of the microprogrammable host machine. Here the <type i> may refer to any one of the previously mentioned types III.1-2 or the primitive type BIT.

Example:

```
TUPLE
   'opcode' OF SEQ 6
   'operand' OF MATRIX 4,4
   'index' OF BIT
ENDTUP
```

As the tuple permits the grouping of different types of objects it therefore is fairly powerful. For example, an object named "flag-status" can be declared as an instance of the type tuple.

```
TUPLE

      'aluflag' OF BIT
      'pcflag' OF SEQ 4
      'progflag' OF SEQ 12

ENDTUP
```

b) Data Objects

In AMPL, there are three main classes of data objects, namely the type, variable and constant data objects. These data objects are also defined by the host machine (except the pseudovariables and literal constants). That is, their names, types, and structures are defined at the microarchitectural level (see section 4.2.2 on machine resources declaration). In fact, apart from the pseudovariables and literal constants, an AMPL microprogrammer can only reference those data objects appeared to him in the machine organization. In other words, only those machine resources already declared in the microarchitecture description (see section 4.2.2) can be referenced. In case there is any reference to pseudo-variable or literal constant data object (will be described later), which must be declared before being referencd. There are basically three reasons to support the adoption of such a policy. The first reason is for the reduction of unnecessary errors and ambiguities as the programmer now has to pay heed to the structure of referenced data objects. Second, for the clarity of the program text. Third, such declarations set up a well-defined "scope" for the data objects declared, thus providing some measure of protection to undeclared data objects. These three reasons are also already well understood in the area of programming methodologies.

The declarations for the three types of data objects namely the type, variable and constant data objects are as follows:

(i) Type Data Object

TYPE '<type-identifier>{;<type-identifier>}' OF <type-structure>

where <type-structure> refers to any one of the types (III.1)-(III.5), or simply, the symbol BIT; and <type identifier> is a programmer-defined name for some structured type.

Example:

TYPE 'register; word' OF SEQ 16

The introduction of the type data objects is for better program readability and programming conveniences.

(ii) Variable Data Object

In AMPL, variable data objects are all host-machine-defined and are treated as predefined global varibles. Sometimes, at an abstract level, however, for convenience purpose we can express some particular actions by assigning a value to some virtual data objects. These virtual objects may also provide a means for communication between statements. In AMPL such objects are termed pseudovariables and they are declared as follows:

PVAR '<pvar-identifier>{;pvar-variable>}' OF <type identifier>

Examples:

1. PVAR 'ovflsamp;alu-status-enable-bit' OF BIT
2. PVAR 'H;L;M;N' OF word

Example 1. declares two pseudovariables named "ovflsamp" and

"alu-status-enable-bit" of the type BIT. Example 2. declares four registers H,L,M,N of the type "word" previously declared having a structure of SEQ 16 (see the example for the type data object).

In addition, variables and pseudovariables can also be renamed using the SYNTO (synonymous to) declaration:

'<identifier 1>' SYNTO '<identifier 2>'

Example:

PVAR 'reg' OF SEQ 4

PVAR 'ovfl' OF BIT

'indexreg' SYNTO 'reg'

'flagreg' SYNTO 'ovfl'

(iii) Constant Data Object

The declaration for constant data objects is:

CONST '<identifier>' OF <number type> (<length>) <value>

where <identifier> is the programmmer or machine-defined name corresponds to the machine-defined constant located in read-only memory elements; <number type> can either take BIN (binary), or OCT (octal), or HEX (hexadecimal), or DEC (decimal); and <length> and <value> are the bit-length and value of the data object in the corresponding number system respectively.

Example:

```
CONST 'zeros' OF DEC (16) 0,
CONST 'plus1' OF OCT (16) 1,
CONST 'sign' OF BIN (8) 10000000;
```

Sometimes, for programming conveniences, literal constants may need to be declared, and their declaration in AMPL is

LITCON '<identifier>' OF <number type> (<length>) <value>

Example:

```
LITCON 'N' OF DEC (16) 1
```

Basically, all data-objects are predefined in the machine resources declaration (see section 4.2.2). And the microprogrammer need only have the machine organization chart at hand to start writing his microprograms.

3.2.3 Operators

The basic operator set for AMPL is depicted in Table I. As shown, they are basically of the keyword type. Eight arithmetic operators are allowed in AMPL and parentheses may be used to explicitly specify precedence. Relational operators are used to compare two data items to determine the relationship existing between them. The execution of a comparison statement results in returning a boolean value carrying either the 1 (true) or 0 (false) value.

In addition to the six basic logical operators which, are applicable to variable and constant data objects, a number of low level, hardware-oriented, special operators have also been introduced. Examples are provided by the two operators LSB and MSB which, when they are applied to a variable, set a 'boolean' variable to the value of the least (LSB) and most (MSB) significant bit of the variable, respectively. For example we could write:

```
CONST 'A' OF BIN (8) 10101010;
'B' EQU LSB A
'C' EQU MSB A
```

The result is that B and C assume the values 0 and 1, respectively.

In microprogram writing it is often necessary to specify a shift operation; to achieve this, four 'shift' operators have been provided, allowing different types of shift: namely,

arithmetic left shift (SHAL), arithmetic right shift (SHAR), logical right shift (SHLR) and logical left shift (SHLL). The structure of a shift operation consists of the shift operator keyword followed, by the item to be shifted, a comma, then the number of places to shift. For example, the result of the statement:

```
SHLR A,4
```

would be the shifting of the variable A four places to the right, entering zeros from the left. Jump and main memory read/write operators are also included in the table for microprogram writing.

Besides the operators stated in Table I, macro-operation can also be declared in AMPL as follows:

```
MACRO '<macro-head>'
  <macro-body consisting of AMPL executional statements,
   see section 3.2.4>
ENDMACRO
```

Thus, using macros, AMPL also provides language extensibilities (although macro within macro is not allowed in AMPL). Numerous examples of macro declarations can be found in the sample emulator listing in Appendix A.

Table I. Basic AMPL operators (to be continued)

a. Arithmetic operators

ADD A,B	/A+B
BADD A,B	/Binary add for A, B
SUB A,B	/A-B
BSUB A,B	/Binary subtract, A-B
MULT A,B	/A multiply B
BMUL A,B	/Binary multiply
DIV A,B	/A divided by B
BDIV A,B	/Binary divide

b. Relational operators

A EQ B	/A=B?
A NE B	/A≠B?
A GT B	/A>B?
A LT B	/A<B?
A GE B	/A≥B?
A LE B	/A≤B?

c. Logic operators

A AND B	
A OR B	
NEG A	/negate A
A EXOR B	/A exclusive OR B

Table I. (contd.)

d. Special and shift

A EQU MSB (B)	/ A=most significant bit of B
A EQU LSB (B)	
SHAL A,n	/ Arithmetic left shift of A by n places
SHAR A,n	
SHLL A,n	
SHRL A,n	

e. Jump

GOTO A	/ A is the destination address (or label, then quotes are required)
ACT 'A'	/ activate procedure A
CALL 'A'	/ call procedure A
RTN	

f. Main memory read and write

LOAD R,A	/ A is the address and R is the register to be loaded with MEM(A)
STOR R,A	/ storing content of register R to MEM(A)

Note: All the above keywords together only provide a basic framework of operators for the final instantiated microprogramming language. And these operators are only valid if

they are supported by the host machine (see section 4.2.2) or declared as a macro function name in the microprogram.

3.2.4 Micro-statements

Like S* [1], there are two main types of microstatements in AMPL, namely the simple and the composite microstatements. The basic executional entities in AMPL are the simple statements which correspond to microoperations (MOPs) in the host machine. To keep with Dasgupta's notion of schema [1], the design objective for AMPL's constructs is to provide a skeleton on which the MOPs of the host machine could be mapped. Thus a person knowing AMPL and reading a microprogram written in AMPL(M) for some machine M could obtain a reasonably good idea as to the workings of the microprogram without knowing too much specifics of M.

(I) Simple microstatements:

To conform with the proposed approach for automatic microcode generation, this type of statement is also the basic statements that correspond to the inter-field description statements (see section 4.2.4) of the host microinstruction. In essense, the simple statements are of three types: assign, function and branching.

a) Assign:

A TO B (A.1)

where A is a source data object (i.e. a variable, a constant data object, or a literal constant) and B is a destination (simple or subscripted) variable.

b) Function:

$$E \{\underline{TO}\ B\} \qquad (A.2)$$

where E is a parenthesized expression of one of the forms

<op> {D1{,D2}}

D1 <op> D2

<macro-op> D1,D2

with <op> being a primitive machine-defined operation; <macro-op> being a declared macro operation identifier; D1, D2 being data objects or literal constants.

Note that the syntax and semantics of expressions are not fully specified in AMPL; their form and meaning are determined during instantiation [1]. This approach thus provides a practical solution to the problems posed by the variability of microarchitectures and yet has the merit of simplicity. There are three basic reasons justifying this instantiation method, and they are (quoting Dasgupta's words):

"For in the context of 'real' systems, it is easily verified empirically that (1) the number of distinct expressions in a given instantiation is quite small, (2) the expressions are quite simple, and (3) the primitive operators in the instantiated language are mostly obtained in the operator set defined in AMPL or are valid compositions of these operators; and yet the weak specification of function statement form allows 'deviant' functions, whose syntax and/or semantics do not easily conform to standard patterns, to be defined without too much difficulty

during instantiation".

c) Branching:

Branching statement may take one of the following forms:

GOTO <destination> (A.3)

or CALL '<procedure identifier>' (A.4)

or ACT '<procedure identifier>' (A.5)

or RTN (A.6)

where <destination> is either a statement label or a data object or an expression E (possibly parenthesized) as defined in the function statement. In case the <destination> refers to a statement label, quotes have to be introduced as those for the <procedure identifier> in statements A.4 and A.5.

The issue on the employment of GOTO statement has been thoroughly discussed in [12] for any further discussion. The reason here for its inclusion may best be argued by Dasgupta's words:

"The inclusion of this statement may appear regressive as a philosophy of language design, and yet from the viewpoint of source program optimization it seemed essential to include it".

The word "ACT" (activate) here refers to the calling of procedure without return.

Finally it is important to note that since simple microstatements correspond to MOPs and the precise action of MOPs are machine-specific, the semantics above are only partial

specifications for simple microstatments. For a given host machine, additional rules may be required to complete their semantics (see section 4.2.4 and 4.2.6). (An example is the setting of the overflow flag for some 'add' operations while leaving the flag intact for other operations).

II) Composite Statements

There are basically four types of composite microstatements in AMPL. Basically, they are composed of simple microstatements strung by the AMPL program reserved words. These composite statements are the AMPL statements consisting the constructs by means of which control structures in AMPL programs are developed.

a) Compound:

The compound statement is basically the type of statement having the construct for denoting micro-parallelism. A compound statement looks like the following:

```
COBEGIN
        S1
        S2
        ..
        ..
ENDCO                (A.7)
```

where Si are simple statements that are all to be executed within the same microcycle. This statement basically specifies the concurrent execution of the Si's. Thus, microinstruction

optimization can therefore be performed at the source level.

b) Condition:

```
IF 'C'
    THEN D
    {ELSE Dn }
ENDIF                                   (A.8)
```

where C is either a machine defined testable condition or a program condition; D and Dn can be one of the statements (A.1-A.7). The "ELSE Dn" included in the curved bracket is optional. 'Program condition' here refers to the program testing condition on pseudo-objects (i.e. pseudovariables or literal constant data objects). Under such a case, D and Dn may also include some operations on pseudo-objects not necessarily conform to A.1-A.7 (see the Appendix on the emulator listing); and normal program expansion like that for the macro occurs.

c) While:

```
WHILE 'C' DO D                          (A.9)
```

where C and D are as described for the 'condition' statement.

d) Repeat:

```
REPEAT D UNTIL 'C'                      (A.10)
```

where C and D are as described for the 'condition' statement.

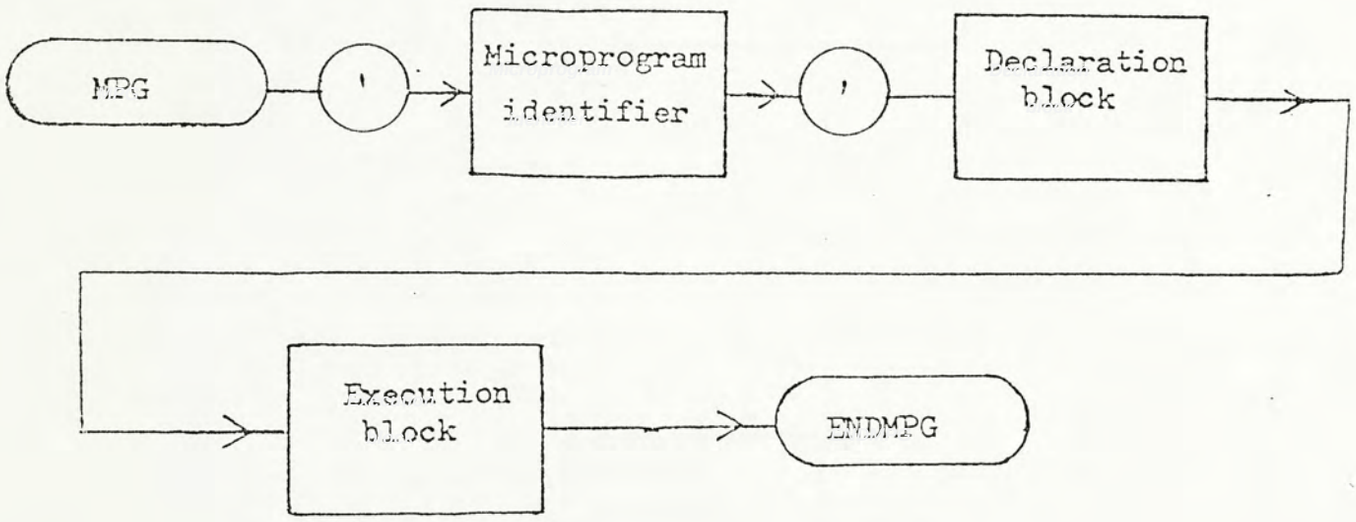
Essentially the composite statements have incorporated the necessary components for writing structured and readable microprograms.

Note: All the underscored keywords appeared in the executional statements are AMPL reserved words with their normal usage as in a macroprocessor. They can be mapped to microoperations only if they are declared in the microarchitecture description (see Chapter 4).

3.3.5 Syntax Summary

The syntax of AMPL is described using the following series of syntax diagrams.

Microprogram :



Declaration block :

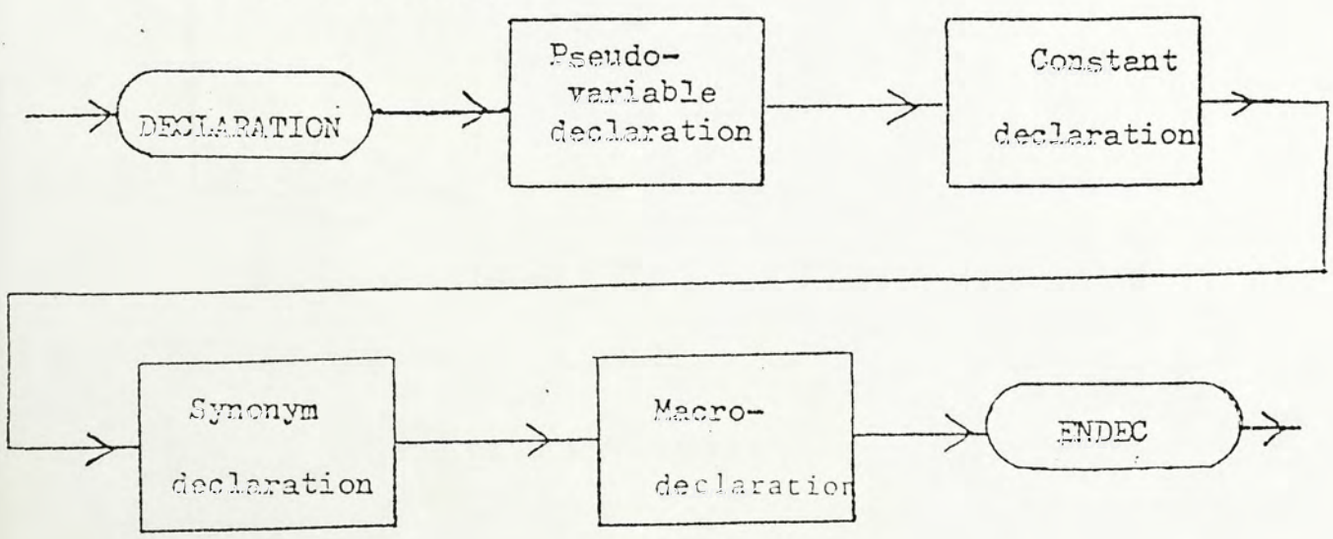
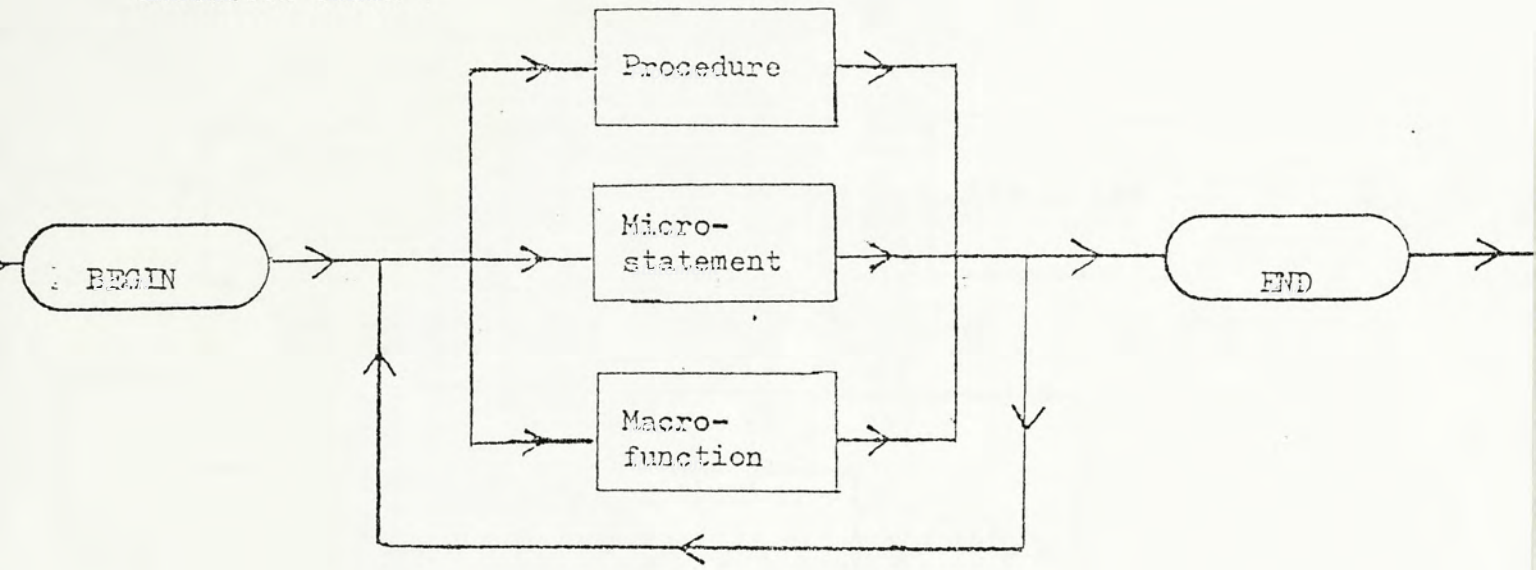


Fig.2.3 AMPL Syntax Diagrams (to be cont'd)

Execution block :



Pseudovariable declaration :

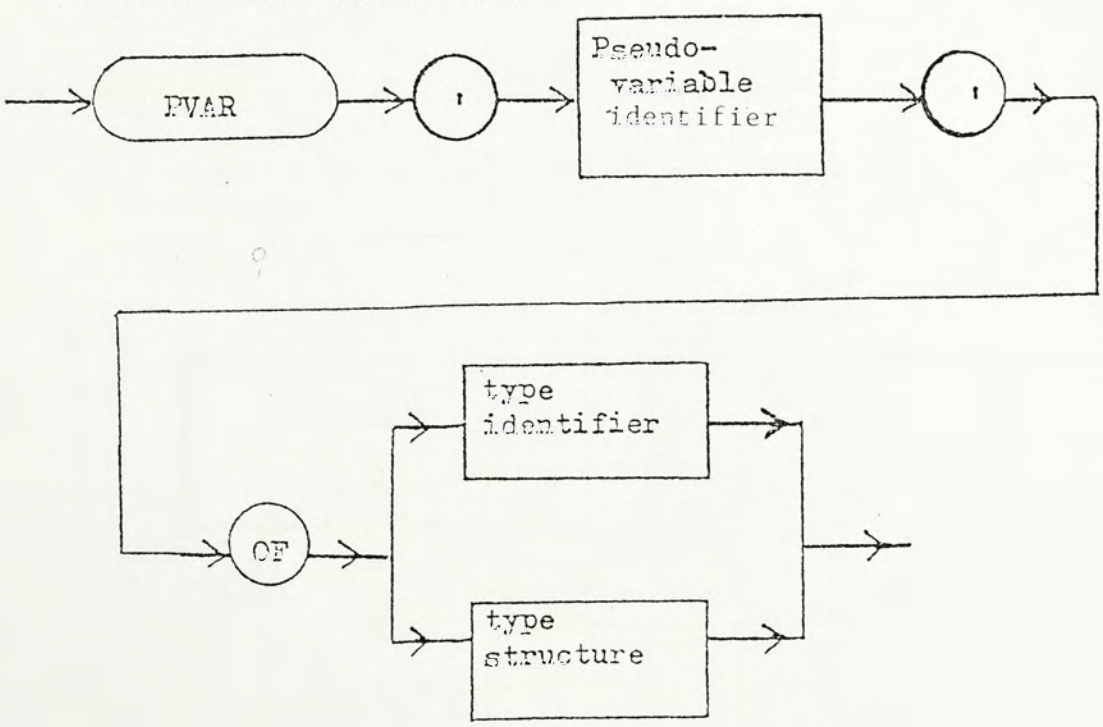
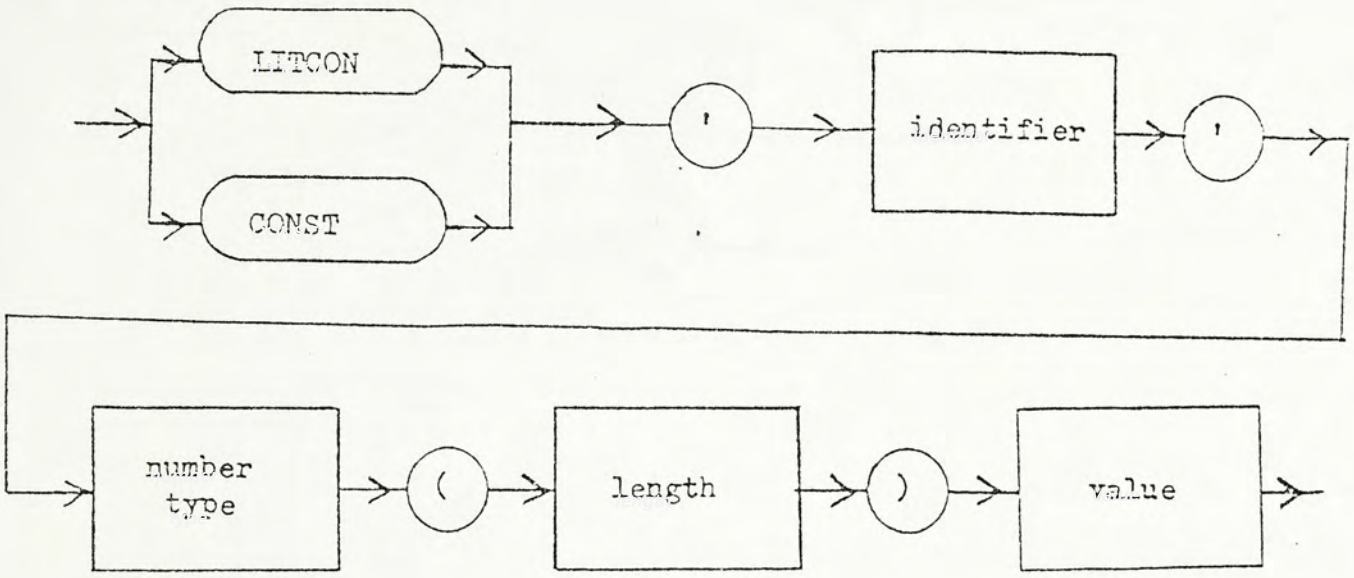


Fig. 3.2 AMPL Syntax Diagrams (cont'd)

Constant declaration :



Synonym declaration :

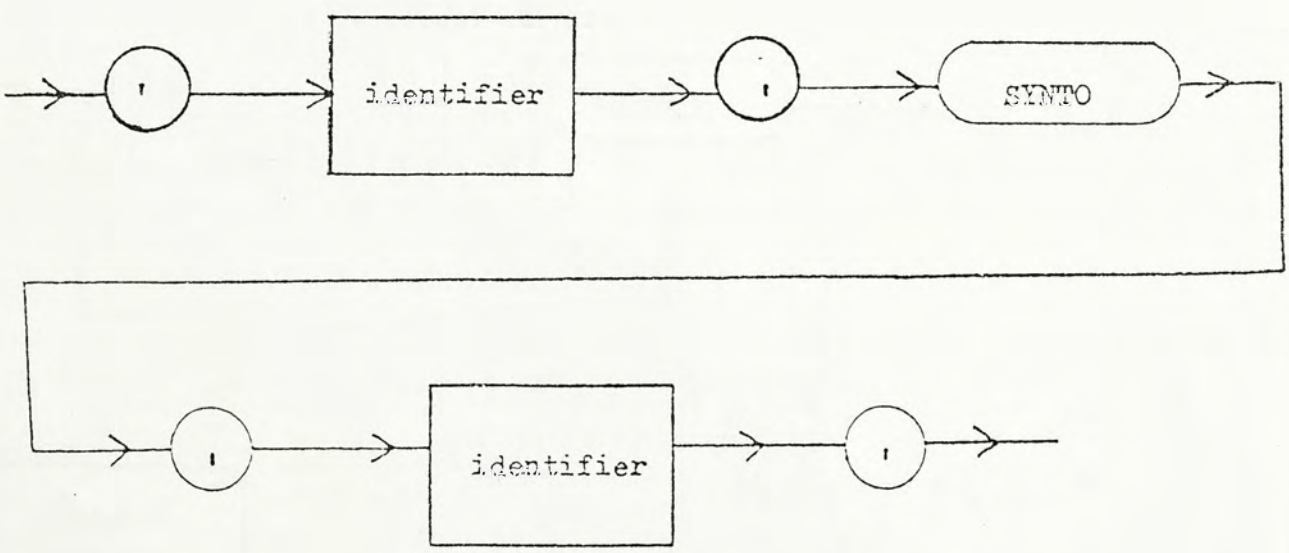
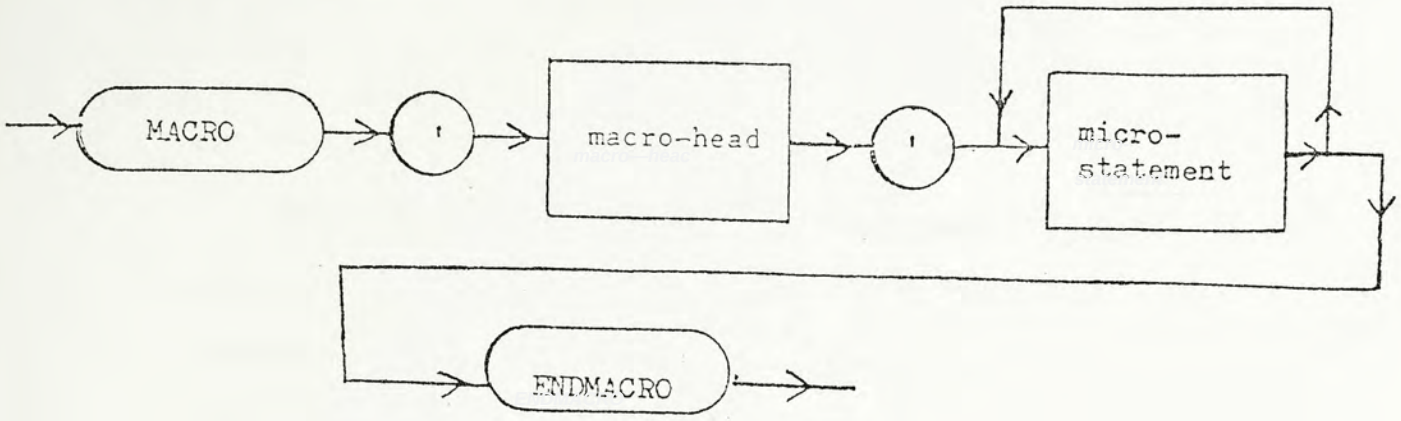
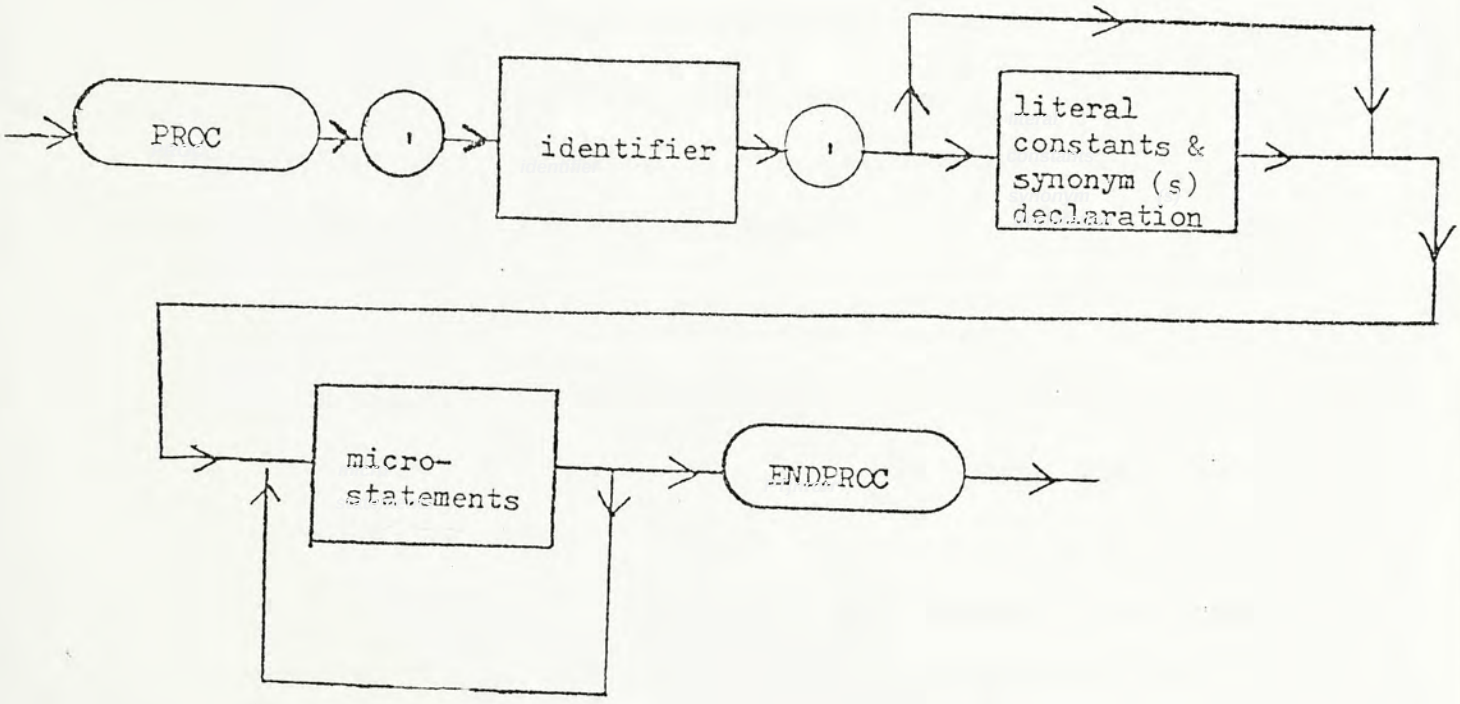


Fig.3.2 AMPL Syntax Diagrams (cont'd)

Macro declaration :



Procedure :



Microstatement(s):

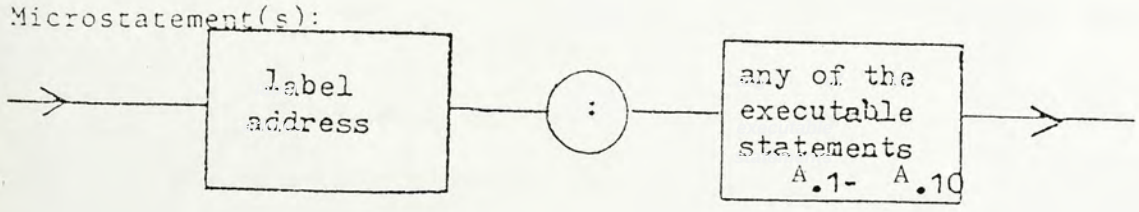


Fig.3.2 AMPL Syntax Diagrams

Chapter 4. Design of MIMOL

4.1 Design Considerations

The second step to realize the proposed approach for automatic generation of microcode is to design or choose a language for microinstruction description. As the research in this area is still sparse [13], and with regard to the uniqueness of the approach, we therefore decided to design the language ourselves. The language we designed is called MIMOL (microinstruction modelling language).

Certain points need to be considered before we go into the design of MIMOL. Basically these points are related to the capabilities as required by a microarchitecture description language, and they are listed as follows:

(i) MIMOL must be capable of describing the details of every field of a microinstruction including

a) the corresponding position of each field in the microinstruction,

b) every controlled microoperations and machine resources corresponding to each field of the microinstruction, and their corresponding microcode values,

c) the corresponding default value of each field, etc.

(ii) MIMOL must be capable of describing the relations between fields of the microinstruction, including

a) how fields are arranged under different formats of the microinstruction,

b) how a field links with the other in the microinstruction, for example, field A acts as the source and field B acts as the destination resource select field for a register transfer operation, etc.

c) the conflict condition between fields, like resource conflict (e.g. field A cannot initiate an operation involving register A while field B is using the same resource, etc.).

(iii) MIMOL must be applicable to different microarchitectures. This is also the basic requirement posed by the retargetability of AMPS.

(iv) MIMOL must be portable. The portability of MIMOL implies that it can be implemented on most of the existing high level languages.

(v) MIMOL has to conform with the approach proposed for the automatic generation of microcode. In other words, inter-field description statements in MIMOL have to be consistent with the structure and syntax of AMPL's simple statements.

We now go into the language MIMOL in details.

4.2 The Language MIMOL

4.2.1 An Overview

MIMOL is basically a declaration language. It contains constructs for the following purposes:

a) Definition of the hardware resources of the machine microprogrammed;

b) Definition of the field encoding details of the microinstructions of the host machine. These details include the field names, default values and positions in the microinstruction, etc.;

c) Definition of relationships between fields of micro-instruction. These relationships basically are established by the operations/operands relations between different fields of the microinstruction;

d) Definition of the machine defined testable conditions that will be assumed during microprogramming the host machine;

e) Definition of implied operations between operations/operands of different fields of a microinstruction.

In addition to allowing users to define the semantics of microinstructions, MIMOL is also designed in such a fashion that it can be applied coherently with AMPL for the generation of microcode. As one of the primary aims of designing AMPS is retargetability, the formalism adopted in MIMOL also allows users to describe microarchitectures in a completely machine-independent manner.

4.2.2 Resource Declaration

The declaration of machine resources of the machine microprogrammed provides a basis for other parts of microarchitecture declaration. All the rest declarations, say, the intra- and inter-field declarations, etc. will also refer to and be limited by the resource set declared. Thus, the resources declared also provide a check list for later declarations. Besides, these declared resources will also be treated as the predefined variable data objects for AMPL programs.

Essentially, all machine resources appeared in the microarchitectural level will all be declared. Resource declaration is basically used to show the existence of particular machine resources in the microarchitecture. Thus only the resources' names and type structures are of importance. The syntax for resource declaration in MIMOL is as follows:

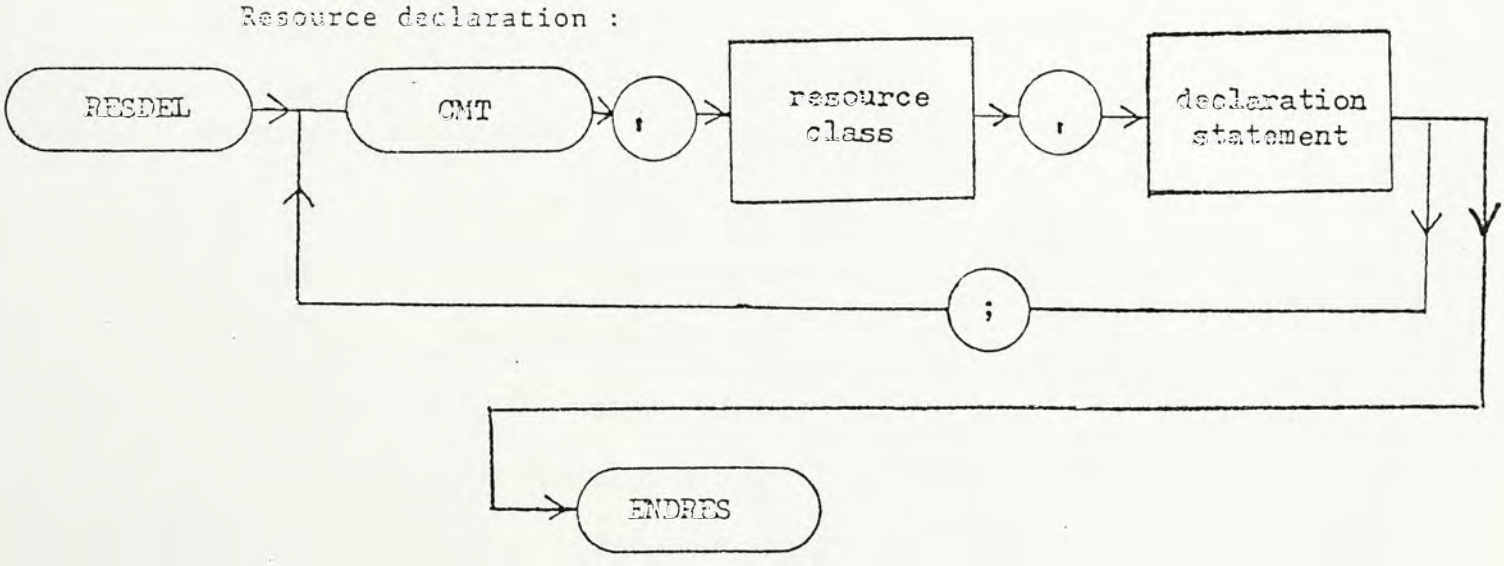


Fig.4.0 Syntax Diagrams for Resource Declaration (cont'd)

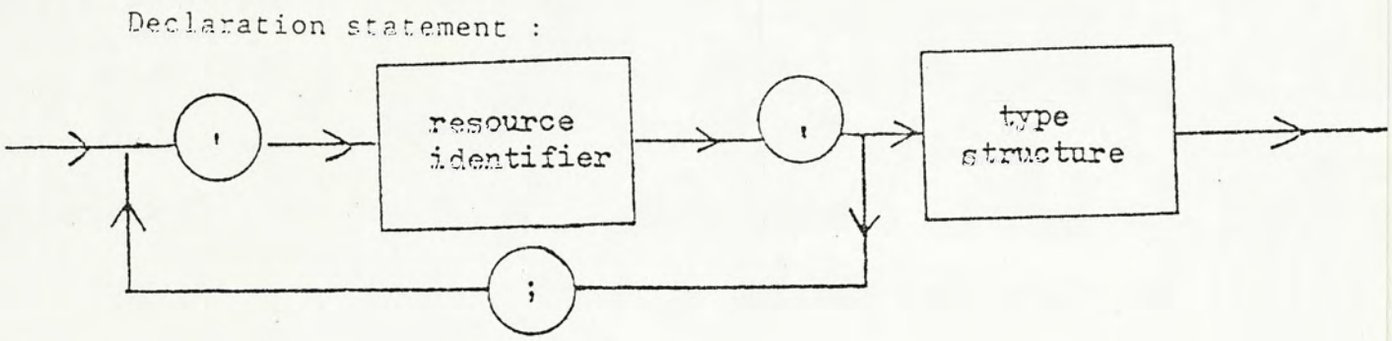


Fig.4.0 Syntax Diagrams for Resource Declaration

where RESDEL and CMT are the keyword commands for resource declaration and comment respectively; "type structure" may either be one of the structures as described in AMPL (section 3.2.2). The "resouce class" appears in the syntax diagram above is arbitrary and introduced by the user solely for his convenience and clarity.

```
Example:    RESDEL
                CMT 'register'
                        'R1;R2' OF SEQ 16
                CMT 'aluports'
                        'APORT1;APORT2' OF SEQ 8
                CMT 'control-store'
                        'CS' OF MATRIX 1023,32
            ENDRES
```

4.2.3 Intra-field Declaration

Intra-field declaration names microinstruction fields and enumerates their constituent microoperations/operands and positions in the microinstruction word. Generally speaking, the bit position(s) of a field in a microinstruction word should be stated in a consecutive manner and the leftmost bit is always the most significant one (see latter example for the Chen's machine). However, fields are allowed to overlap (as will be later shown). And fields are basically associated with the resource named "CS" (i.e. control-store) declared at the start of the declaration.

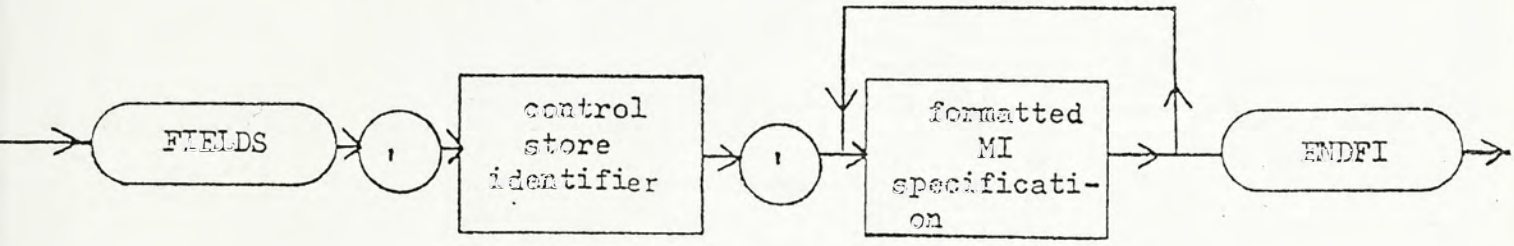
Intra-field declaration is basically of two steps: the identification of microinstruction field names and relevant bit position(s) in the microinstruction; the enumeration of each of these fields details.

a) 1st step: Identification of microinstruction fields

A microinstruction (hereinafter called MI) may take different formats. In other words, we may have $MI = MIF_1 | MIF_2 | MIF_3 | \ldots MIF_n$ or $MI \in \{MIFi \mid i=1 \text{ to } n\}$. Within these MIs of different formats, certain fields are format affected while certain fields are not. Thus, by grouping these format unaffected fields into sections, we can save our effort in later declaration by simply including these sections. The step of microinstruction fields identification is essentially depicted by the following syntax diagrams. As will be illustrated, the MIMOL keywords (capitalized words) are already self-explanatory. The keywords "AT" and "ATX"

mean "at the position(s)" and "at the positions extending from" respectively.

Field identification :



Formatted MI specification :

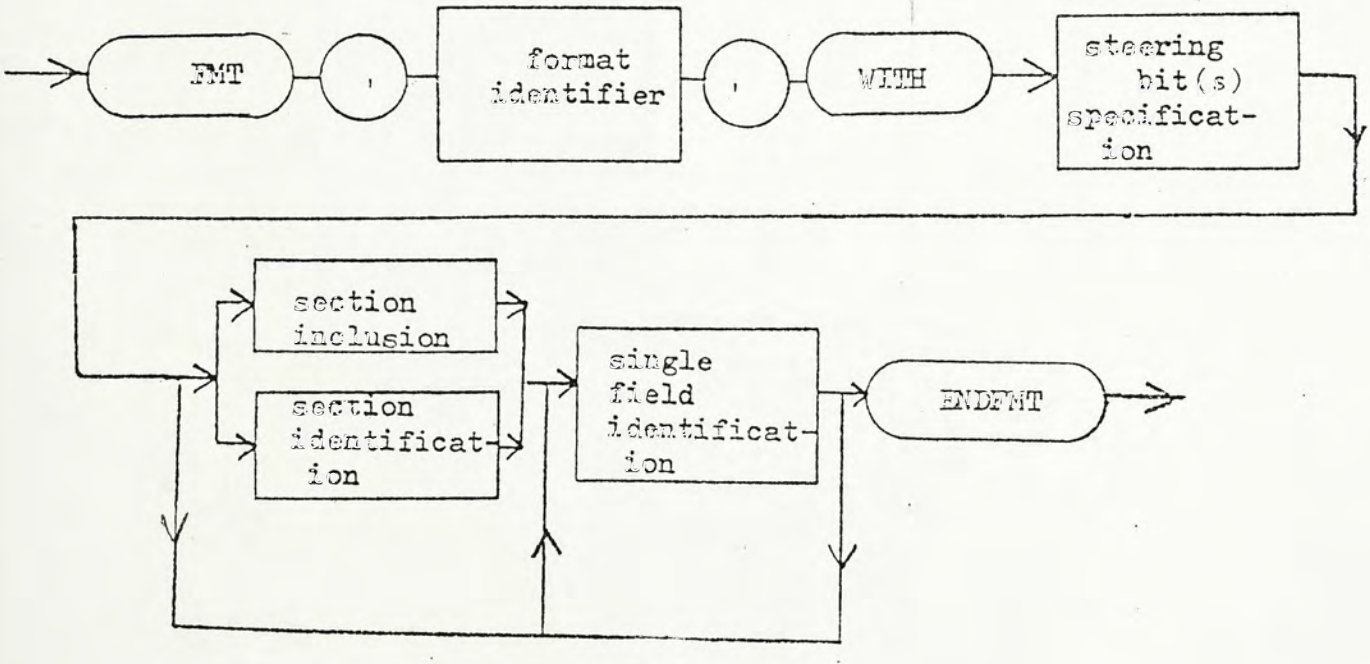
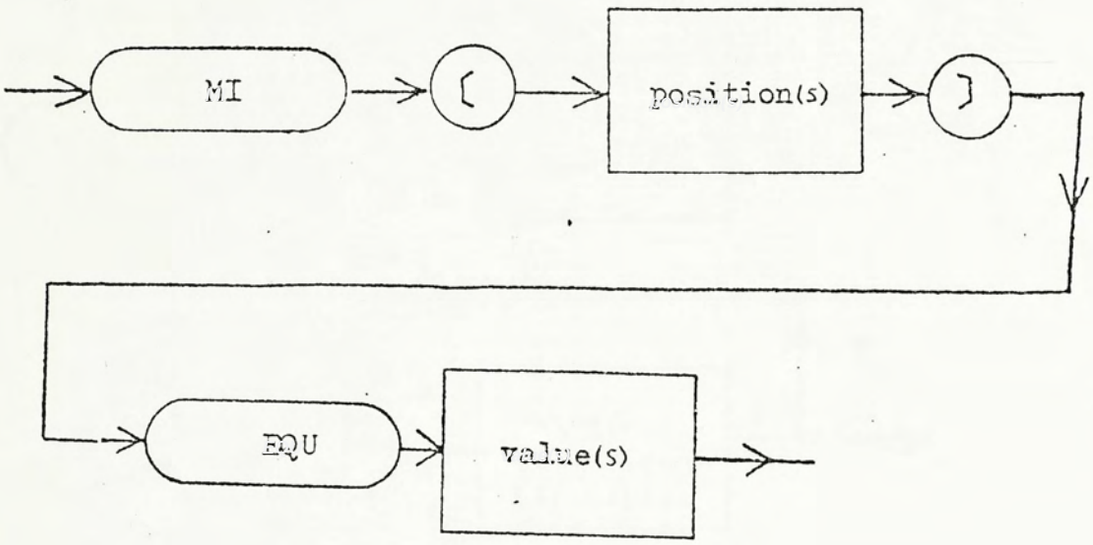


Fig.4.1 Syntax Diagrams for Field Identification (cont'd)

Steering bit(s) specification :



Section inclusion :

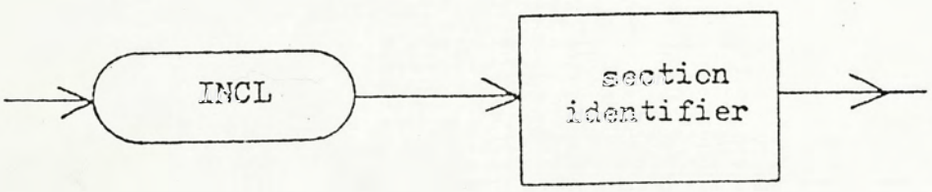
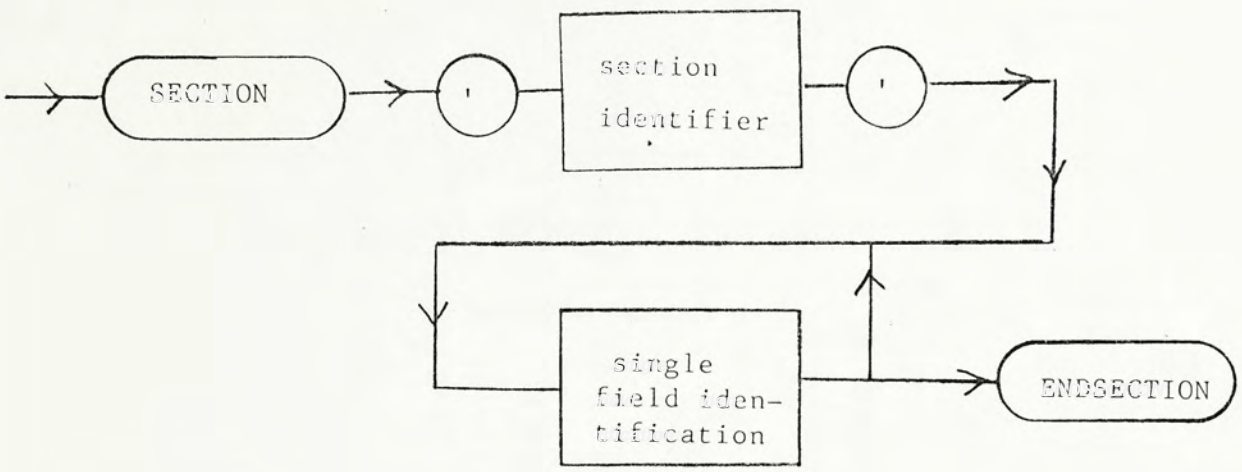
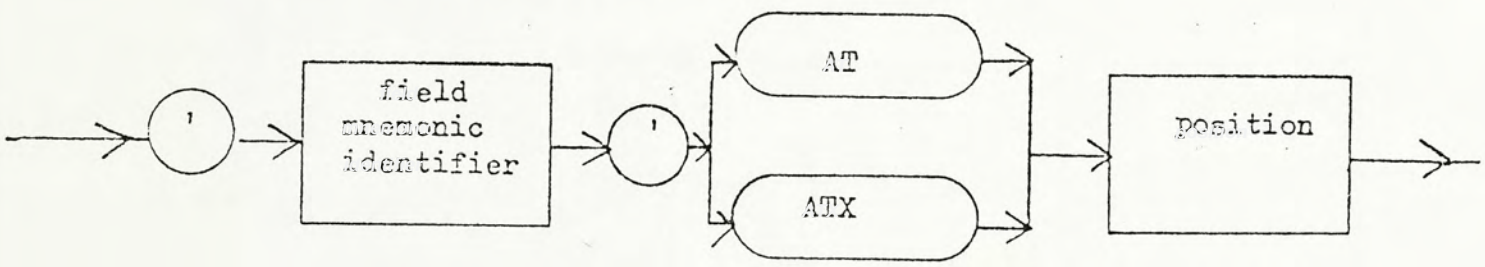


Fig. 4.1 Syntax Diagrams for Field Identification (cont'd)

Section identification :



Single field identification :



Position(s) :

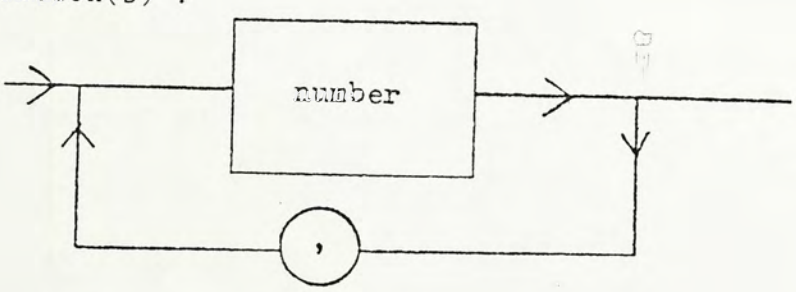


Fig.4.1 Syntax Diagrams for Field Identification (cont'd)

values(s) :

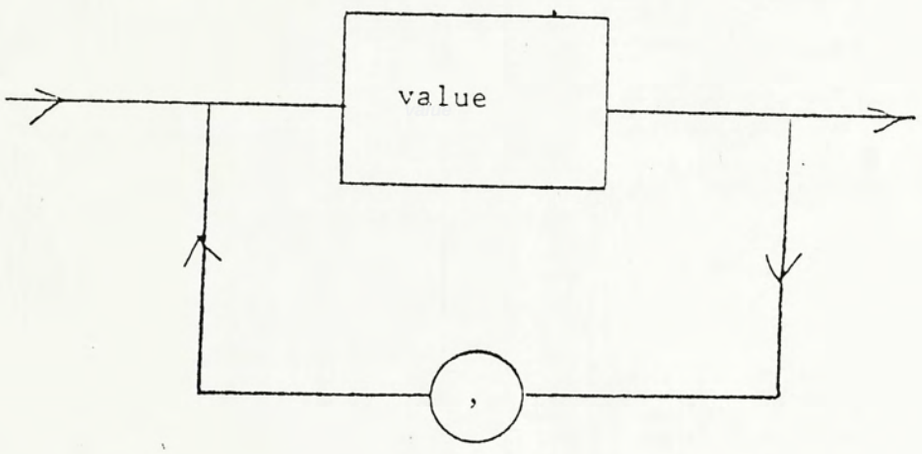


Fig.4.1 Syntax Diagrams for Field Identification

Example:

```
FIELDS  'CS'
    FMT 'f1' WITH MI[0] EQU 0  /declare a MI of format named f1
                               and set the format decision
                               position (MI[0]) into 0
     SECTION 's1'              /declare a section named s1
     'ainput' AT 48,49
     'binput' AT 43,44
      ..
      ..
     ENDSECTION                /end of section declaration
     'index' AT 15,16
     'flag' ATX 17,20
```

```
    ..
    ..
  FMT 'f2' WITH MI[0] EQU 1
    INCL s1                    /include section s1
    'condition' AT 23,24       .
    ..
    ..
ENDFI
```

b) 2nd step: Field enumeration

Although there is a wide spectrum of available microarchitectures, MI fields, however can be classified as the following three basic types: the operation select, the operand select, and the emit fields; where "emit" field means that the field acts as a data supply field. The enumeration of fields' details can be depicted by the following syntax diagrams.

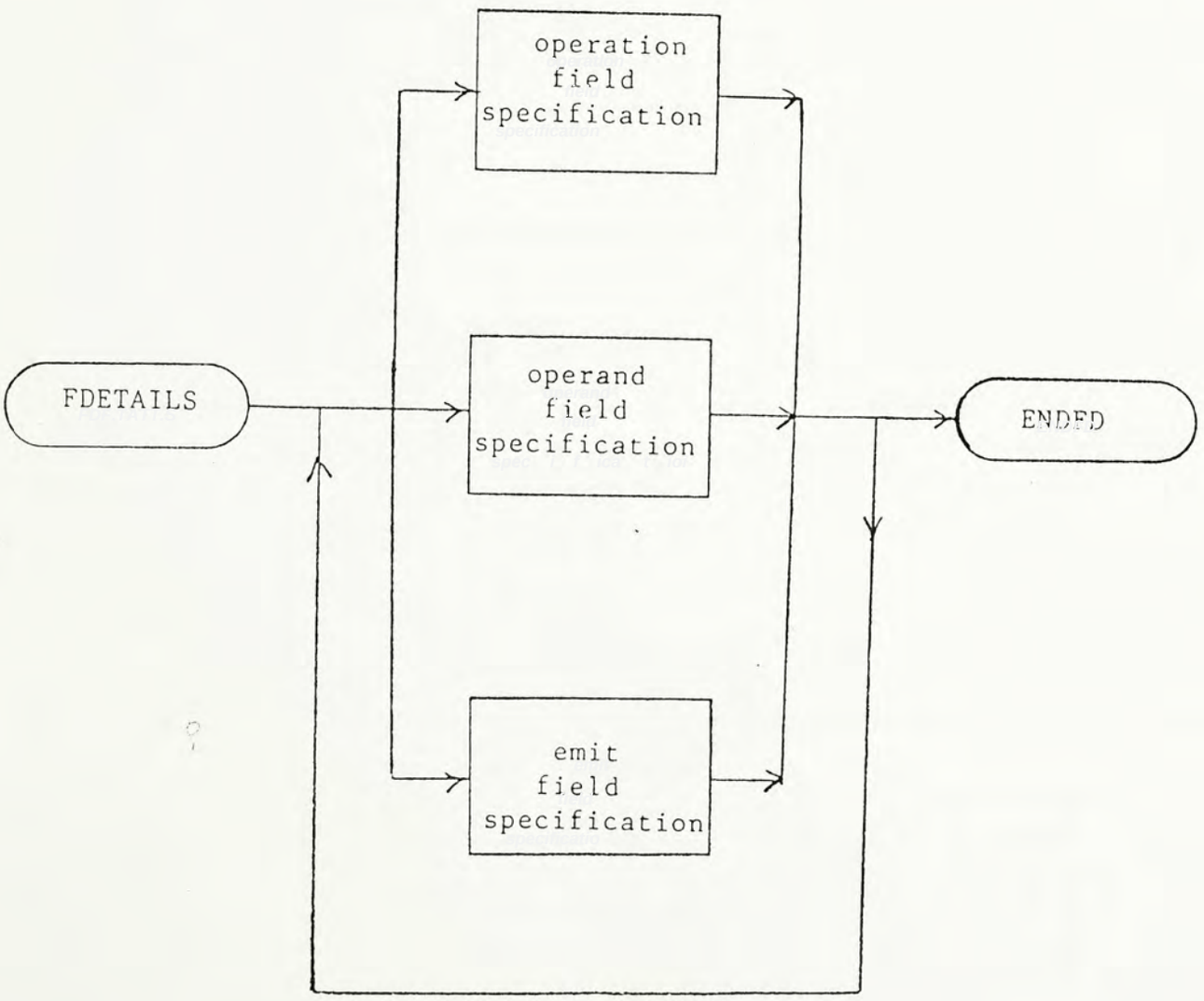


Fig.4.2 Syntax Diagrams for Field Enumerations (cont'd)

Operation field specification :

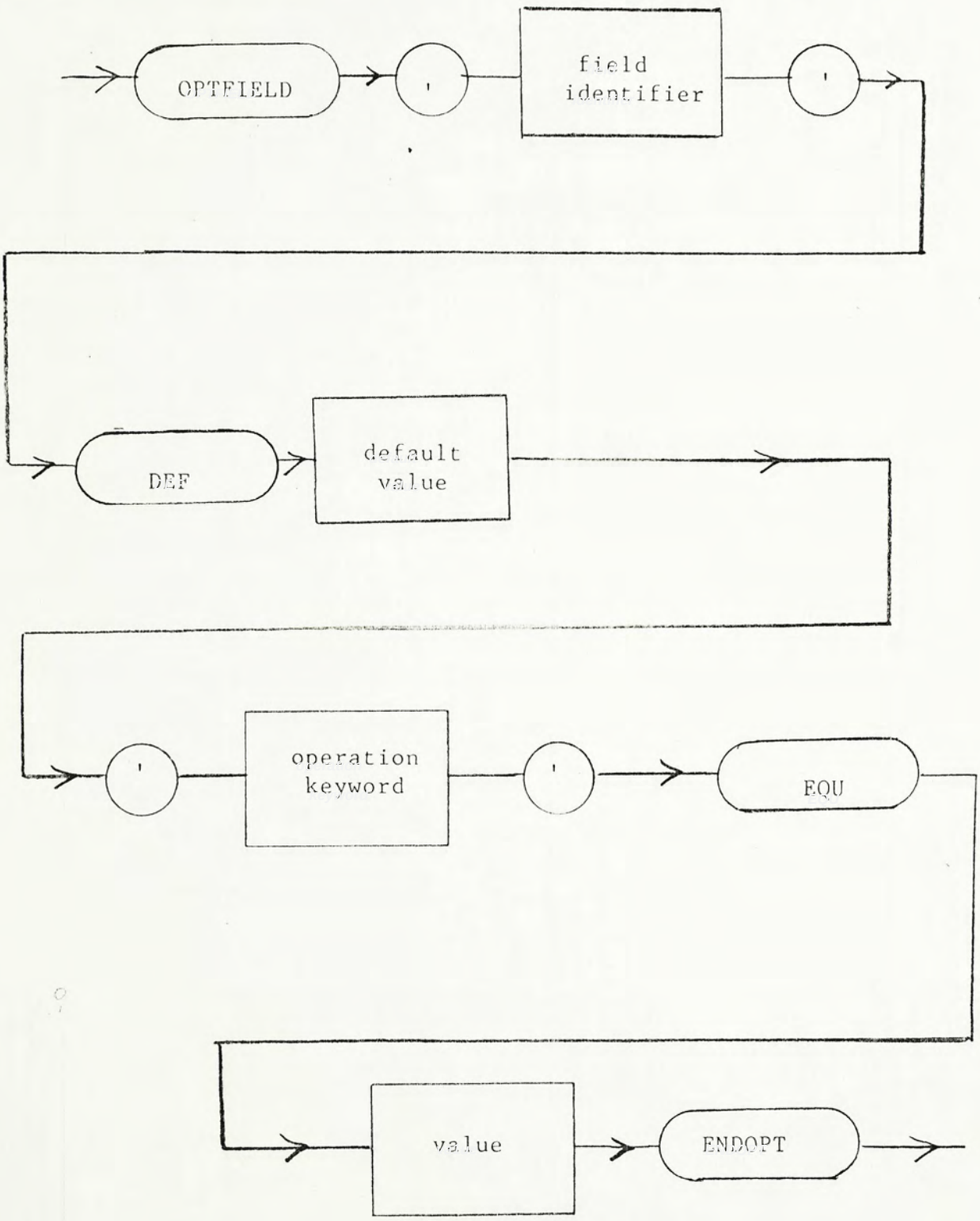


Fig.4.2 Syntax Diagrams for Field Enumerations (cont'd)

Operation field specification :

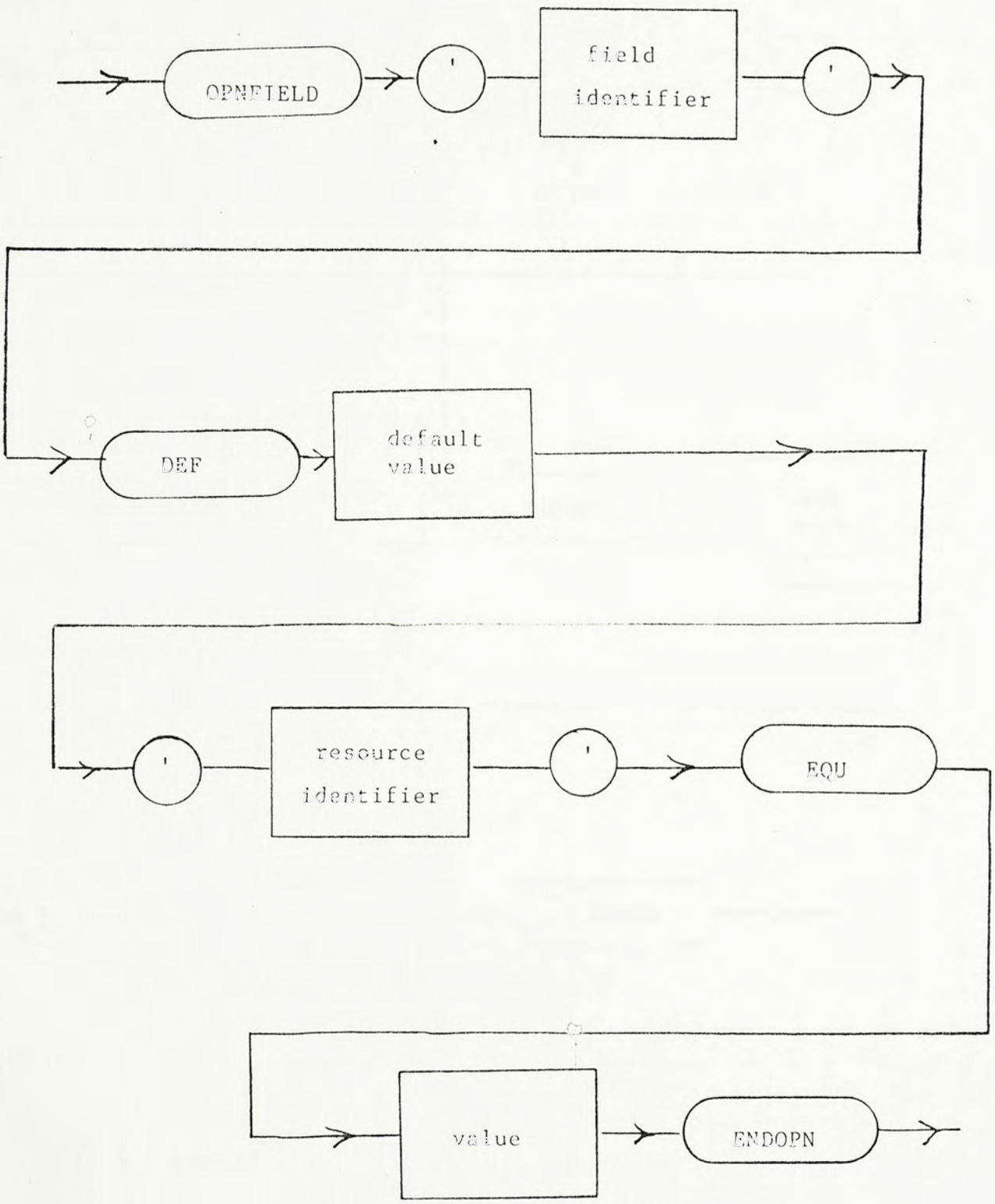


Fig.4.2 Syntax Diagrams for Field Enumeration (cont'd)

Emit field specification :

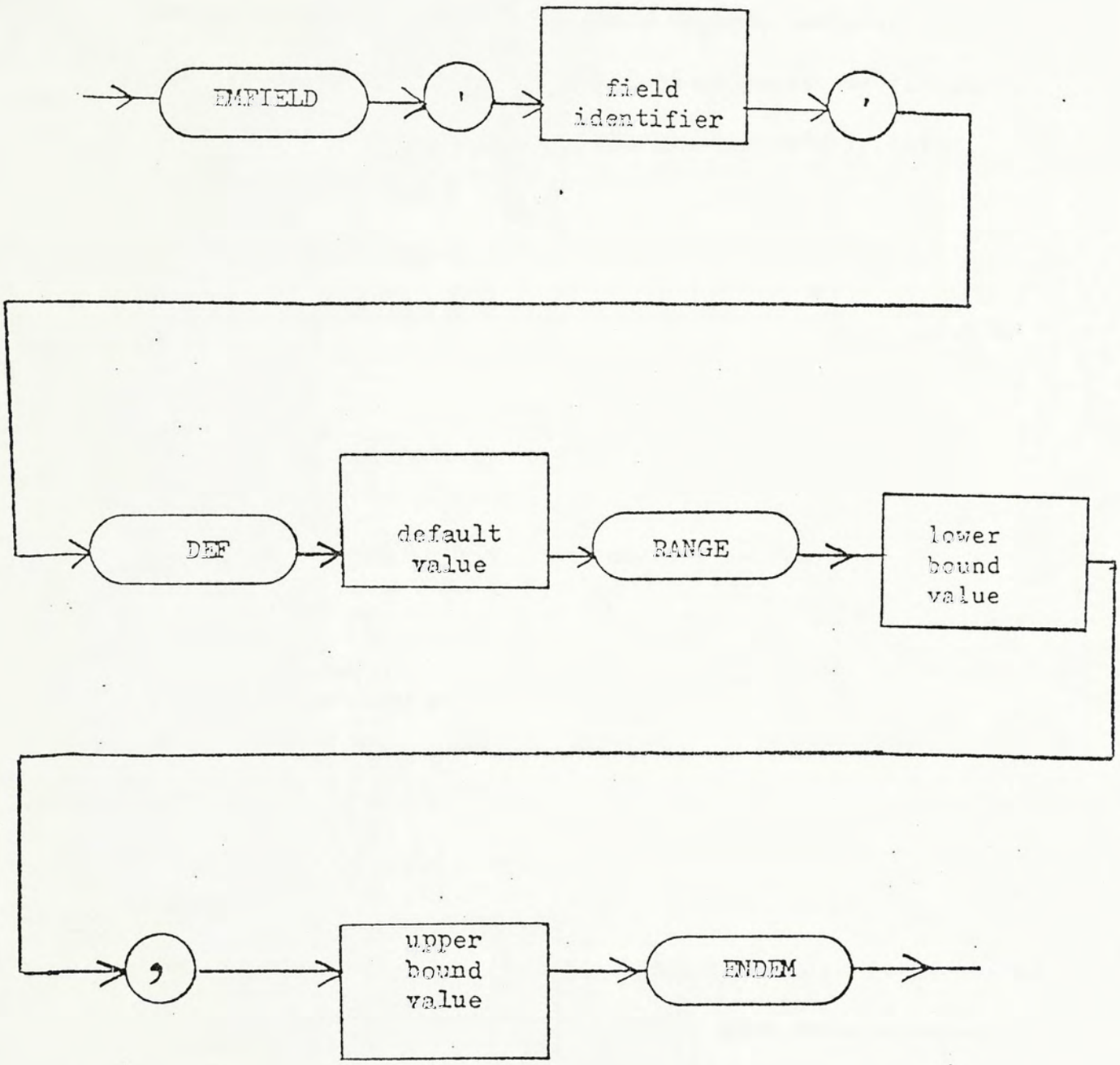


Fig.4.2 Syntax Diagrams for Field Enumerations

Example:

```
   FDTAILS                          /field details declaration
     OPTFIELD 'aluop' DEF 0         /declare an operation field named
                                     "aluop" with default value 0
              'nop' EQU 0
              'add' EQU 1
              'sub' EQU 2
                 ...
                 ...
     OPTEND
     OPNFIELD 'ainput' DEF 0        /declare an operand field named
                                     "ainput" with default value 0
              'R1' EQU 0
              'R2' EQU 1
                ...
                ...
     OPNEND
     EMITFD  'emit'  DEF  0         /declare an emit  field  named
                                     "emit" with default value 0
        RANGE 0,255                 /with range between 0 to 255
                                     (2's complement are assumed
                                     for -ve value)
     EMEND
ENDFD
```

4.2.4 Inter-field Declaration

Inter-field relation basically depends on the operation-operand or register-transfer relations exhibits as accorded to the use of the microinstruction fields. To conform with the notion of a common core syntax under the proposed approach for automatic microcode generation (see section 2.3), inter-field description statements in MIMOL, therefore, are also similar to AMPL's basic statements in their syntactic appearances. Besides, this similarities further allows users to have a coherent picture between the MIMOL described microinstruction and the 'would-be' microprogramming language developed from AMPL.

Inter-field relations in MIMOL are essentially depicted by the following types of syntactic statements:

(i) Assign:

<operand field identifier> TO <operand field identifier>

(B.1)

(ii) Function:

E {TO <operand field identifier>} (B.2)

where E is an parenthesized expression of one of the forms

<operation field identifier> {<emit or operand field identifier>, {<emit or operand field identifier>}}

or

<emit or operand field identifier> <operation field identifier> <emit or operand field identifier>

(iii) Branching:

GOTO <emit or operand field identifier>{,<emit or operand field identifier>}

(B.3)

or CALL <emit or operand field identifier> (B.4)

Although only three primitive syntactic statements are available to describe the inter-field relations, it is believed that they are sufficient for the purpose as all MOPs are basically primitive (assign, branching and function) actions. In cases where the MI consists of multiple such relations (e.g. a MI requires two or more 'function' statements to describe its inter-field relation), the occurrence specification is introduced. The syntax for occurrence specification has the form

OCCUR <occurrence number>

Essentially, the occurrence specification is apppended at the end of each of the above syntactic statements (B.1-4) during use. Its presence at a particular statement is to indicate the number of occurrences of that type of statement in the inter-field declaration. For the 'ASSIGN' statement, if the number of occurrence is 1 or the destination field referenced is the same as its predecessor in the statement, this occurrence specification can be omitted. For the 'FUNCTION' statement, if there is no destination field referenced the occurrence specification can also be omitted.

In essense, the provided syntactic statements for inter-field relation description give an abstract representation of the

possible primitive operations initiated by the MI. By comparing B.1-B.4 with A.1-A.4, one can readily observes that they are actually identical in their syntactic outlook.

(Note: The "ACT" and "RTN" are not included in MIMOL for inter-field description. This is because "ACT" is in fact a variant of GOTO and "RTN" is an implied operation of "CALL". While "ACT" and "RTN" may exist in the microprogram for better program structuring, they are not necessary for the inter-field description purposes).

4.2.5 Machine Defined Testable Conditions Declaration

In MIMOL, the syntax for the declaration of a machine defined testable condition is as follows:

```
CONDITION

  EQUOP '<operation identifier> <declared resource>{,<delcared
                                or operand field    resource or
                                  identifier        operand field
                                                    identifier>}'

ENDCON
```

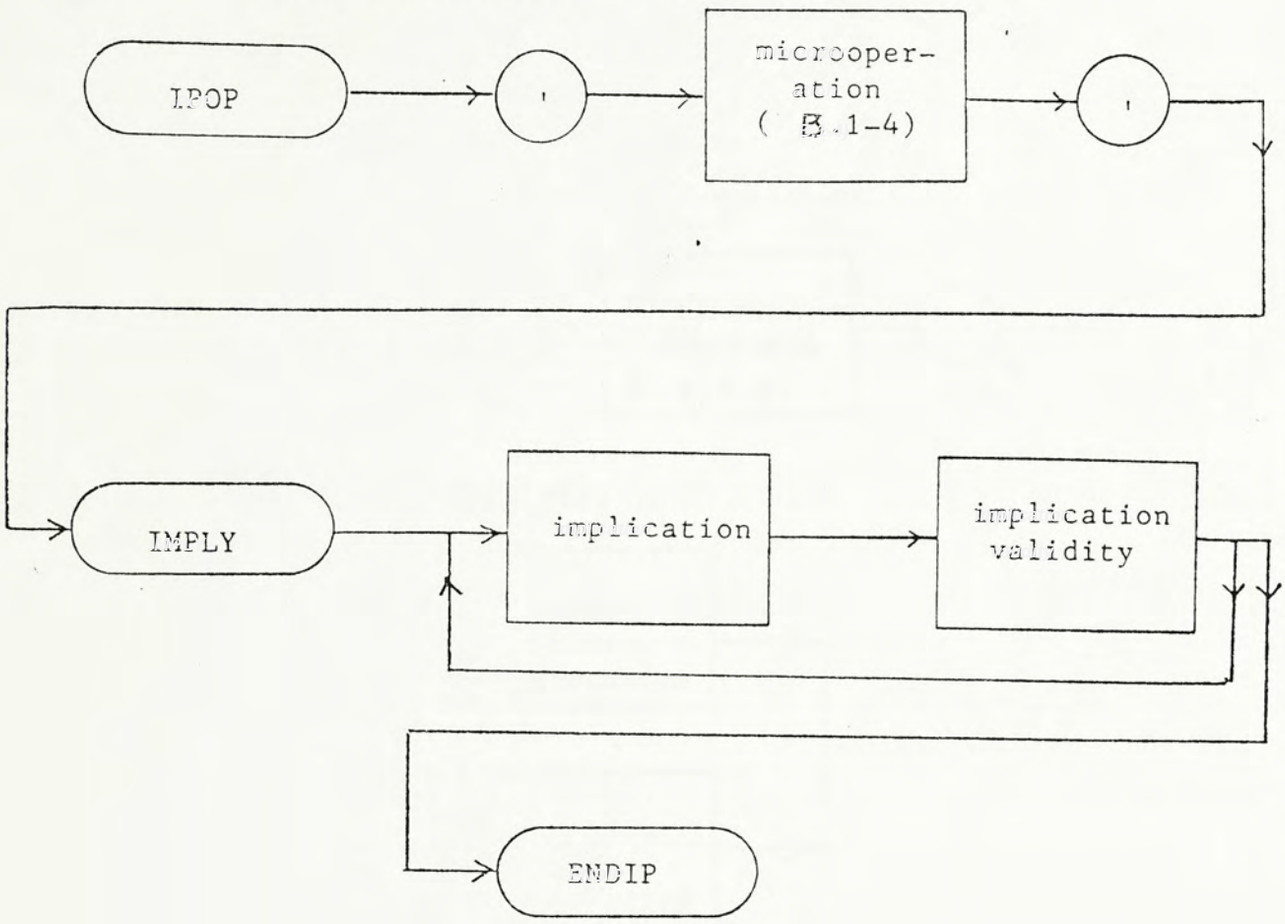
Essentially, defining a machine defined testable condition is equivalent to pinpointing a particular micro-operation. This is because the set-up of a specific machine defined testable condition is always associated with the setting of certain bits of the MI. This in turn means that a certain microoperation is initiated. In other words, setting up a machine testable condition is equivalent to a function operation statement (B.2). Therefore the keyword 'EQUOP' (equivalent operation) is used.

These declared machine conditions will then be the only valid conditions referred to in microprograms.

4.2.6 Implied Operations Declaration:

This declaration refers to those cases where some microoperations must be or shouldn't be executed inclusively or exclusively at some microinstruction cycles due to the resource utilization or timing requirements of the microarchitecture. These declared implications then provide rules for proper relation between different operations initiated by different fields of the MI. The syntax diagrams for the implied operations declaration are as follows:

Implied Operations Declaration :



Implication:

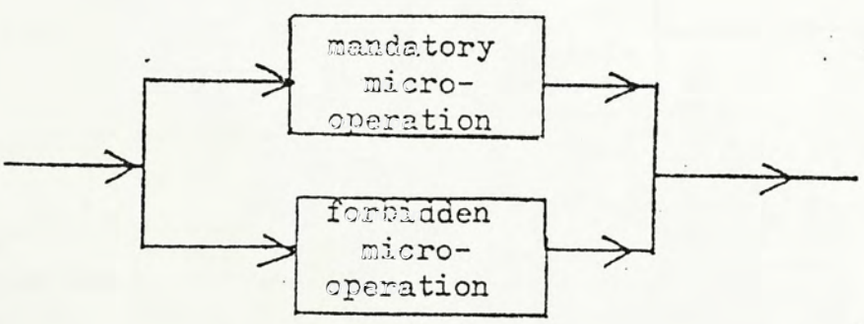
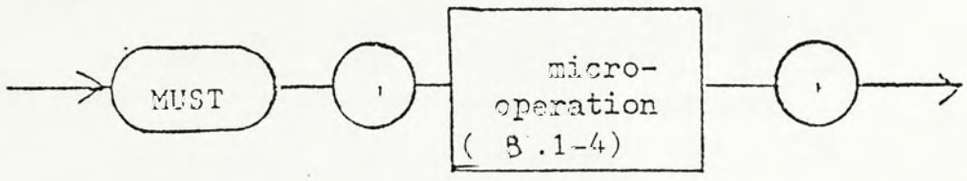
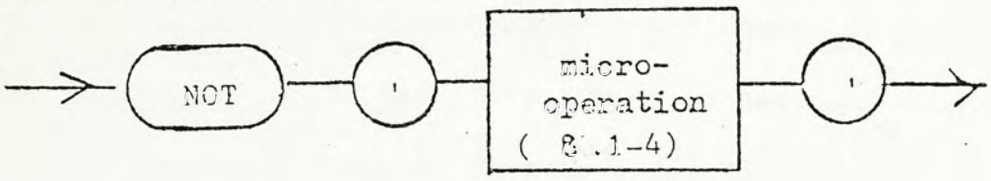


Fig.4.3 Syntax Diagrams for Implied Operation Declaration (cont'd)

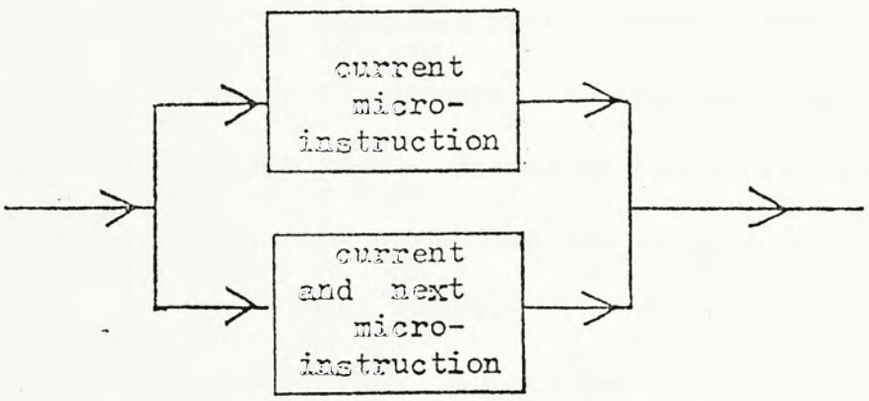
Mandatory microoperation :



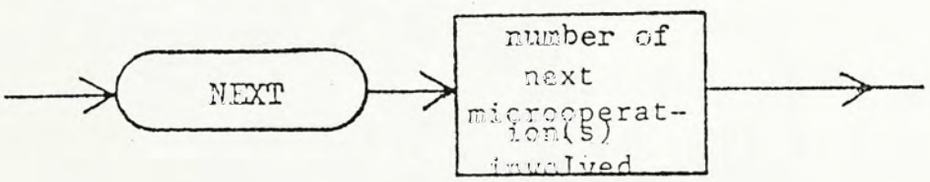
Forbidden microoperation :



Implication validity :



Current and next microoperations :



Current microinstruction :

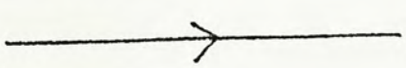


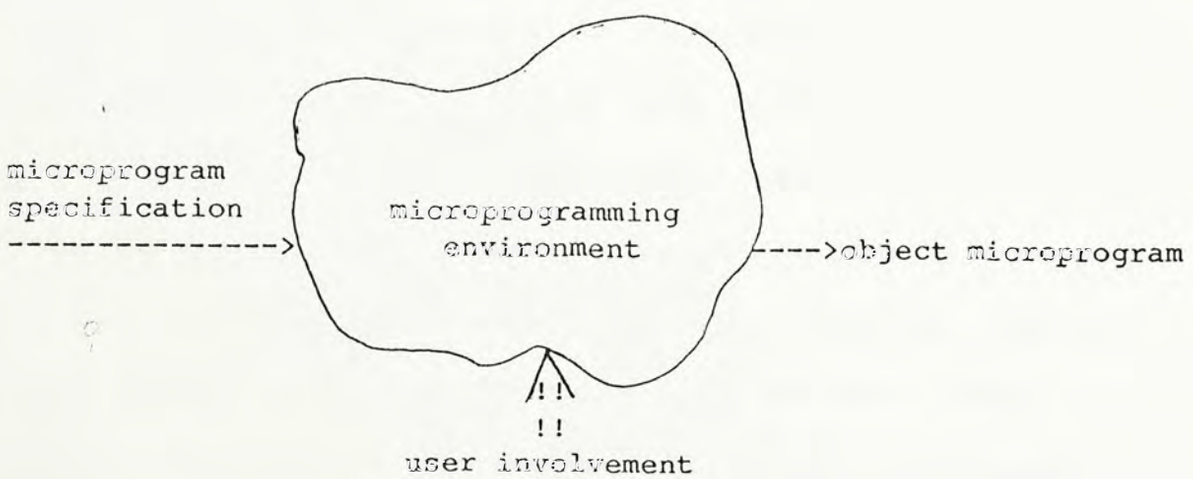
Fig.4.3 Syntax Diageams for Implied Operation Declaration

Chapter 5. The VS/APL-based Microprogramming Environment

5.1 What is a microprogramming environment?

Before discussing the different aspects of a microprogramming environment, we must first address the central question of what a microprogramming environment is. However, this is indeed difficult and risky for there has been only relatively little organized research in this area; not just to mention the definition of the term. Thus, 'microprogramming environment' may sound and mean quite dissimilarly to different people.

Microprogramming environment, as its name suggests, is the environment under which the process of transforming microprogram specification into machine executable microprogram (or called object microprogram) takes place. The relationship between the microprogramming environment and its user, is illustrated below.



In the broadest sense, a microprogramming environment

includes the technical methods, the management procedures, the computing equipment, the software (including the supporting tools), and the physical workspace involved in the development of the final object microprogram. And the software involved for the object microprogram development may include the following:

a) the microprogramming language;

b) the microinstruction description language;

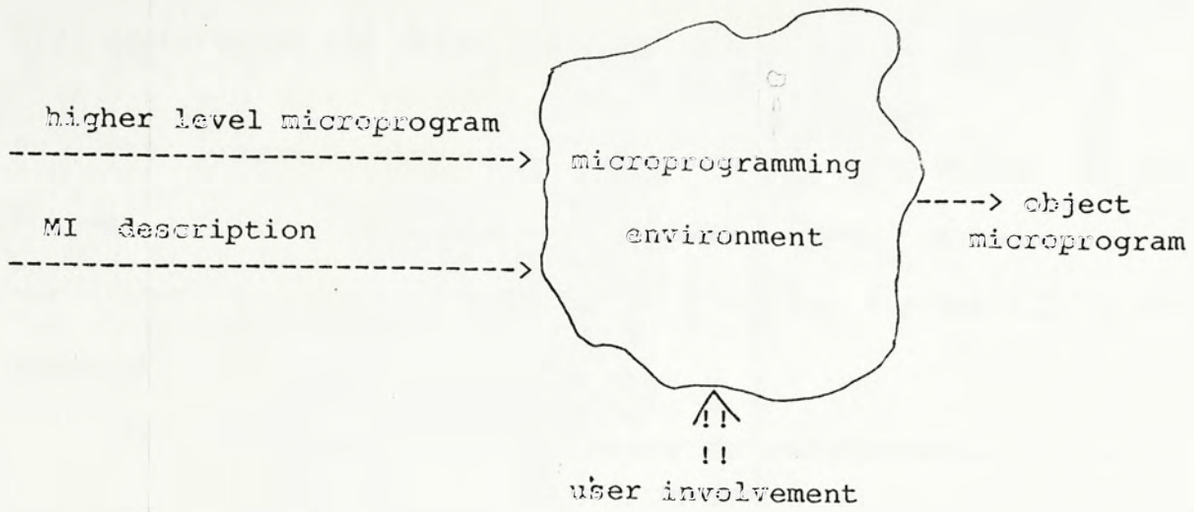
c) translators for the above two languages;

d) the microprogram editor;

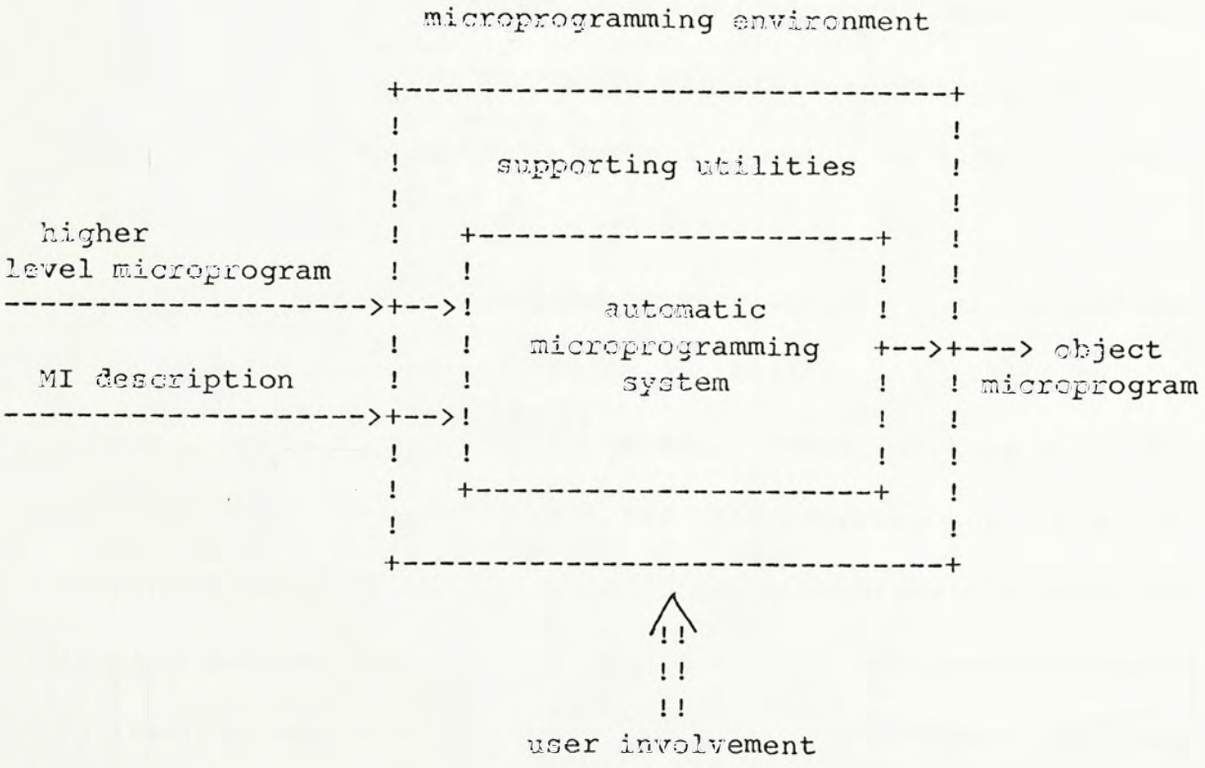
e) the microprogram verifier and/or simulator.

It should be noted that items d) and e) are not shown as the components of an automatic microprogramming system in Fig. 2.1. This is because microprogram editor, verifier and/or simulator are basically the supporting utilities for the automatic microprogramming system rather than its parts.

Here, as far as our interest is concerned, we will restrict our discussion to the supporting tools or machine software provided by a microprogramming environment. Also, when we talk of 'microprogramming environment', we are referring to the supporting software that the microprogrammer might use in the course of preparing his object microprogram. In other words, our discussion scope for microprogramming environment is as follows.



Thus the relation between the microprogramming environment and the automatic microprogramming system may be depicted as below.



5.2 Features of the VS/APL-based Microprogramming Environment

The language chosen here for the implementation of the automatic microprogramming system is VS/APL. (The original specification of APL is contained in [14], and for details of the language, please refer to [15]).

VS/APL provides an environment for microprogramming which has the following features:

a) Interactive capability in the specification and design of microprograms;

b) Interactive capability in the definition and specification of machine microarchitecture;

c) Direct executability of AMPL and MIMOL keyword commands results in fast and efficient microcode production. This is easily achieved by declaring these keyword commands as VS/APL function names.

d) Good isolation can be provided between different microprogramming project machines' data. This is basically achieved by the provision of workspaces in VS/APL. These workspaces also provide good isolated areas for the storing of different machines' information. Updating of these machines' information is also made easier and less error prone as the problems caused by the overlapping of variable names between different machines' information are eliminated. Thus, the provided workspaces also improved information security against unintentional erasure.

e) A multi-user environment can be enjoyed to help to develop

microcode. As most computers can only safely support one user during microcode development and testing, VS/APL provides a mechanism whereby a class of programmers can work about microprogramming and computer architecture without tying up a machine in an inefficient manner.

f) Integrity in the development of firmware, as a complete microprogramming environment can be set up with one common language, namely VS/APL.

g) Provision of utilities to help the development of microcode. For example, the provided editor can be used for editing source microprograms and microarchitecture description. Microcode development is also benefited through the use (directly or indirectly) of the abundant available system funtions (e.g. ⎕FX, ⎕CR, ⎕LC, etc.). Finally, the microcode produced can also be stored up as a variable in the VS/APL system and thus saves later "re-translation" of the source microprograms.

Chapter 6. An AMPS Prototype for the Chen's Machine

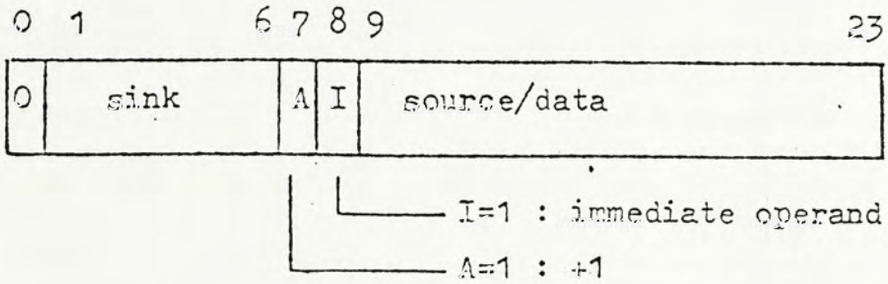
To demonstrate the feasibility of the design presented in the previous chapters, an AMPS prototype for the Chen's machine [2] has been implemented. The following sections will only highlight the main points of the implementation. Interested readers, may, however, refer to the Appendice and [2] and [3] for details.

To start, it will be very helpful to have a look at the MI and the machine organization of the Chen's machine first.

6.1 Microinstruction and Machine Organization

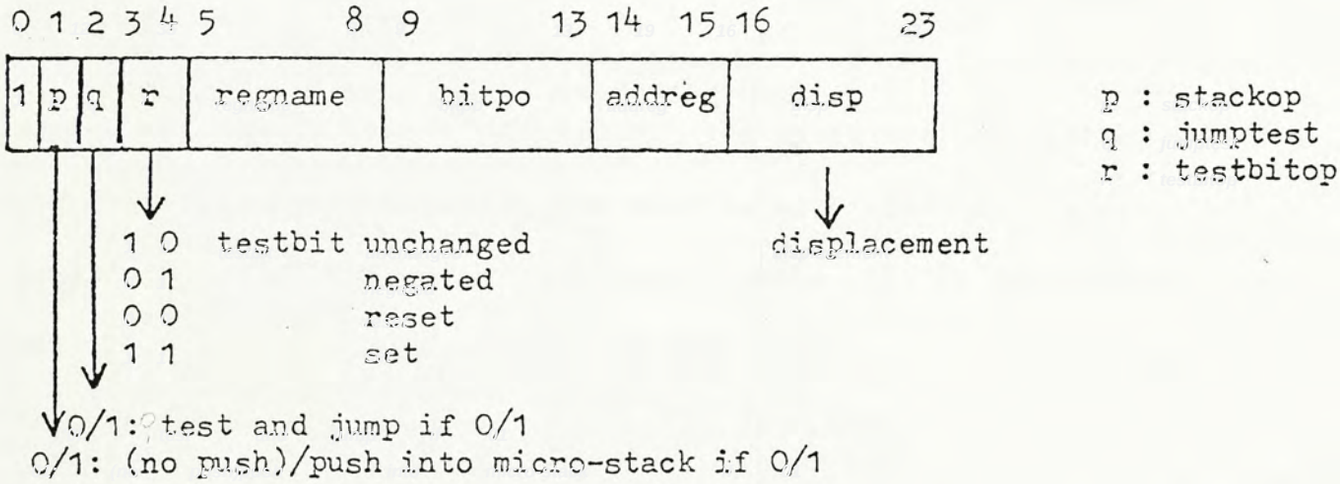
The Chen's microinstructin essentially is of two types: the register transfer and the branching microinstructions. They are shown in Fig.6.1.

Register-transfer microinstruction :



A,I=0,0	C(source) → C(sink)
A,I=1,0	C(source) + 1 → C(sink)
A,I=0,1	data → C(sink)
A,I=1,1	data + 1 → C(sink)

Branch microinstruction :



addreg field contains the address to one of the four micro-index registers, MXR(0) to MXR(3).

effective address = disp OR (MXR(addreg))

Fig.6.1 The Chen's Machine MI

As shown, the microinstructions specify only register transfer and branch operations; arithmetic and logic operations are not included. Yet the semantics of the microoperations are complicated by the fact that the content of microaddress registers MXR(0) is hardwired to a constant zero and that for MXR(1), its most significant bit is hardwired to one. Another complication added into the semantics of the microoperations is the availability of the "unmatched" transfer (will be illustrated later). Below is an illustration to the operation semantics of the Chen's microinstructions.

Referring to Fig.6.1, the distinction between a branch and a register transfer microinstruction lies in the logic state of the most significant bit. If this bit is 1, the instruction word is a branch one, otherwise, it is for data movement among registers.

During a register transfer operation, if the sink register is shorter than the source register, the extra leftmost bits of the data from the source register will be truncated. On the other hand, if the sink register is longer, zeros will be appended to the left of the most significant bit of the source data, as illustrated in Fig.6.2.

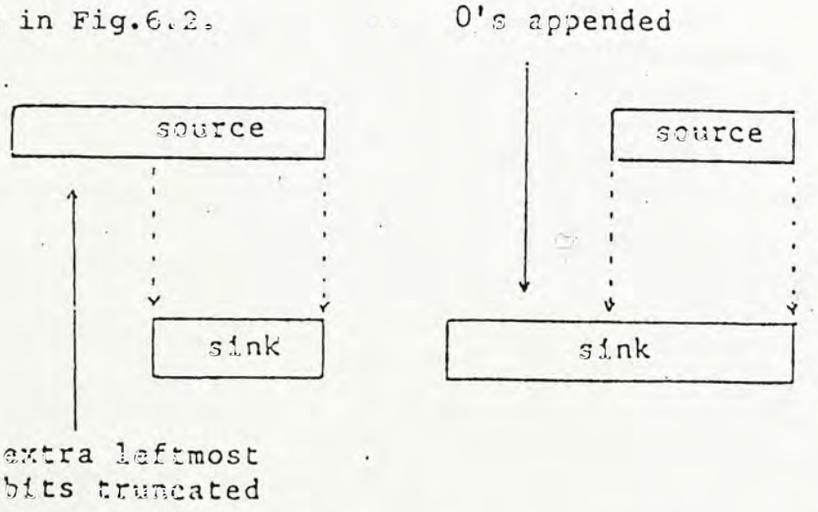


Fig.6.2 An illustration of the 'unmatched' transfer

The same adjustment to the length of the sink register also applies when immediate operand is the source. If the immediate operand or the contents from a source register is incremented by 1 before being loaded into the sink register, it is the result after the incrementation to which this adjustment to the length of the sink register applies.

The execution of a branch microinstruction involves a sequence of microoperations. First a specified condition will be tested (i.e. a machine defined testable condition is set up). This specified condition is the logic state of any one bit of a certain register (counted from left to right in an increasing order) defined by the 'addreg' and 'disp' fields of the microinstruction respectively. If the register under test is longer than 32 bits, only the most signigicant 32 bits are involved in the test. If the 'regname' field contains a zero, an unconditional branch will be made.If the test condition is not satified, no branch is made. Otherwise a branch is made to the effective address calculated using the information contained in the 'addreg' and 'disp' fields (Fig.6.1). The 'addreg' field contains the address to one of the four micro-index register (MXR0-3) in the Chen's machine organization (Fig.6.3). Contents of the 'disp' field is the displacement. The displacement is OR-ed with the micro-index addressed by the 'addreg' field to produce the effective address. But before this effective address is placed into the control store address register (MMAR), original contents of MPC may need to be pushed into the micro-stack according to the logic state of the 'stackop' field in the

branch microinstruction word. A one in the 'stackop' field specifies such a push into the micro-stack if a branch is successfully made.

Bits 3 and 4 of a branch microinstruction are used to specify how the test bit is to be changed after test. Bit patterns of bits 3 and 4 for different actions on the test bit are given in Fig.6.1. Whenever a branch microinstruction is executed, the test bit will be changed according to the specification in bits 3 and 4, irrespective of whether a branch is successfully made or not. Thus a special use of a branch microinstruction is to change the logical state of a certain bit of a certain register. Assume that bit 7 of the register A is known to be in logical state 1. Consider a branch microinstruction that tests bit 7 of register A and will effect a branch if the test bit is in logical state zero (bit 2 of the branch microinstruction =0). The execution of this branch microinstruction will not cause a branch. But by properly setting up the action pattern in bits 3 and 4, the test bit can be changed accordingly.

As shown, arithmetic and logic operations are not included. Yet arithmetic and logic operations can still be implicitly specified for the arithmetic and logic units (ALUs) in the Chen's machine are designed as autonomous units. An ALU as an autonoumous functional unit will contain within itself the control logic necessary for its operations. It communicates with other parts of the system through a set of interface registers.

The way to activate its operations is to pass to it the operands and an operation control word through the interface registers. After it has completed the specified operation with the input operands, it returns the result again through the interface registers. Thus instead of explicitly including arithmetic and logic operations in the specification of the operations within a system, direct register transfers can be used to move the related operands and operation control words to the autonomously functioning ALU to bring about the required operations. In fact this is the basic design philosophy for the components in the Chen's machine. As shown in the organization diagram for the Chen's machine in Fig.6.3, the ALUs, register files and memories in the machine are all autonomous.

The target CPU in the Chen's machine basically has an architecture close to that of the IBM S/360 (see section 6.4). Addresses, lengths, and functions of the registers in the machine are given in Fig.6.4. Another list of the registers in order by address, together with their addresses (as searched by the MI) are also shown in Fig.6.5.

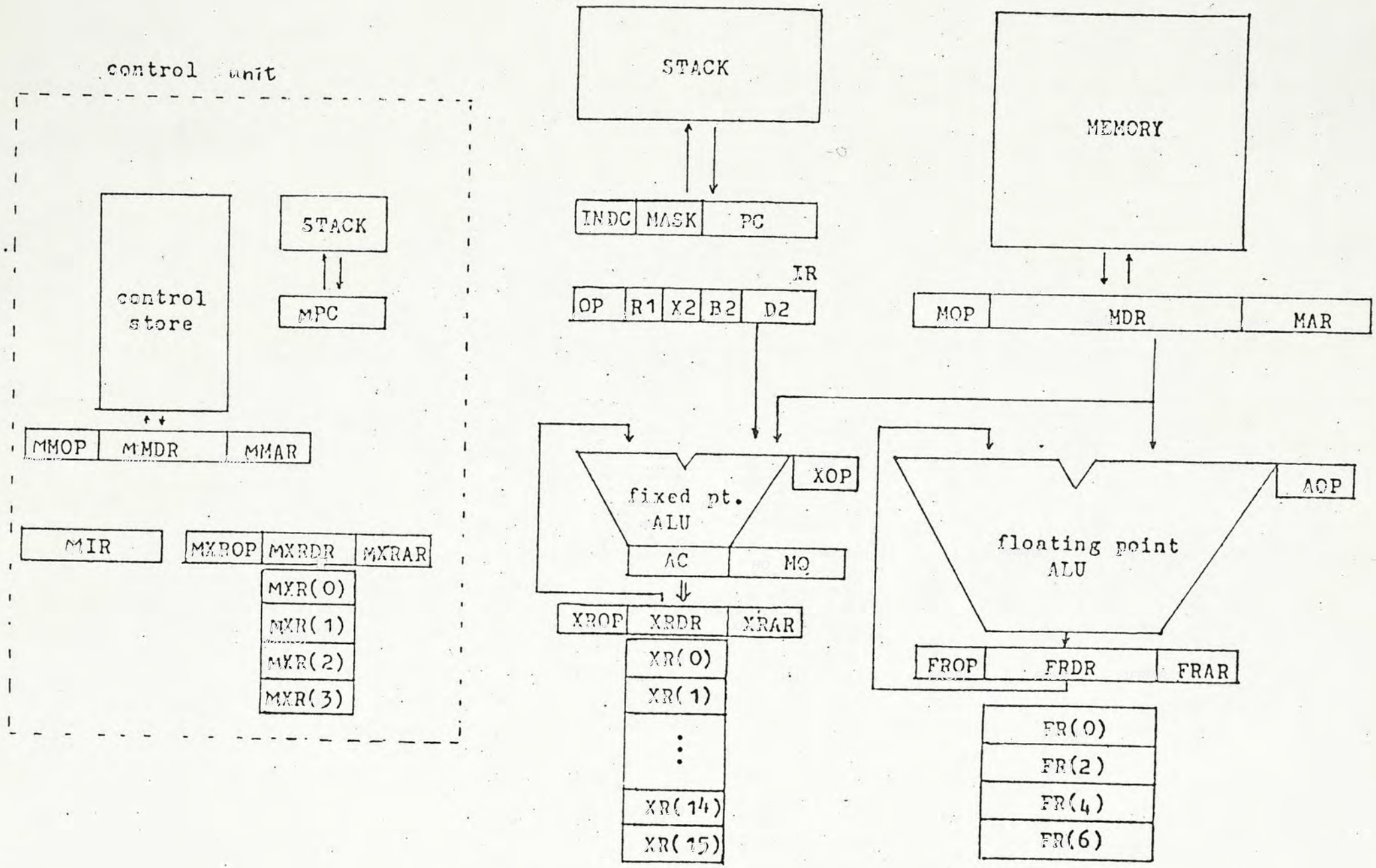


Fig.6.3 Organization of the Chen's Machine

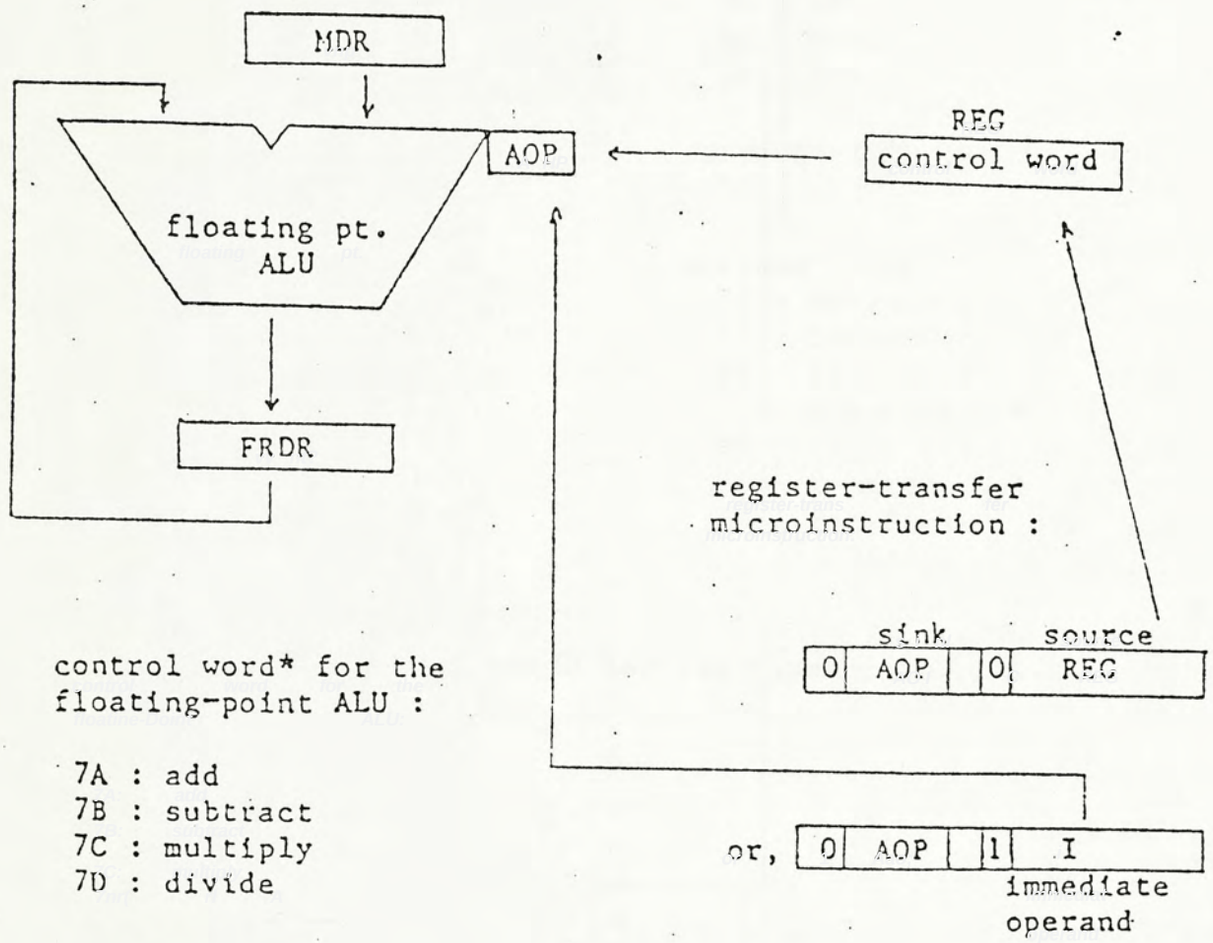
Address (hexa-decimal)	Name	Length(bits, decimal)	Function
08	IR	32	instruction register
1C	IR/OP	8	subregister OP of IR(bits 0-7)
1D	IR/R1	8	subregsiter R1 of IR(bits 8-11)
1B	IR/X2	4	subregister X2 of IR(bits 12-15)
19	IR/B2	4	subregister B2 of IR(bits 16-19)
20	IR/D2	12	subregister D2 of IR(bits 20-31)
0D	MDR	32	memory data register
0F	MAR	24	memory address register
04	MOP	8	interface operation register for memory
05	STACK	40	stack register
0E	PC	24	program counter
09	INDC	8	machine state indicator
0A	MASK	8	interrupt mask
	FR(0)-FR(6)	64	floating-point work registers
0B	FRDR	64	interface data register for FR's
1E	UFRDR	32	subregister UFRDR of FRDR(bits 0-31)
1F	LFRDR	32	subregister LFRDR of FRDR(bits 31-63)
10	FRAR	8	interface address register for FR's
01	FROP	8	interface operation register for FR's
	XR(0)-XR(15)	32	general purpose registers
0C	XRDR	32	interface data register for XR's
11	XRAR	8	interface address register for XR's
03	XROP	8	interface operation register for XR's
00	AOP	8	interface operation register for the floating-point ALU
02	XOP	8	interface operation register for the fixed-point ALU
12	MIR	24	micro-instruction register
13	MPC	16	micro-program counter
15	MMDR	24	control store data register
14	MMAR	16	control store address register
07	MMOP	8	interface operation register for the control store
	MXR(0)-MXR(3)	16	micro-index registers
16	MXRDR	16	interface data register for the MXR's
17	MXRAR	8	interface address register for the MXR's
06	MXROP	8	interface operation register for the MXR's

Fig.6.4 Addresses, names, and functions of the Chen's machine registers. Most significant bit of a register is bit 0.

Register	Address (hexadecimal)
AOP	00
FROP	01
XOP	02
XROP	03
MOP	04
STACK	05
MXROP	06
MMOP	07
IR	08
INDC	09
MASK	0A
FRDR	0B
XRDR	0C
MDR	0D
PC	0E
MAR	0F
FRAR	10
XRAR	11
MIR	12
MPC	13
MMAR	14
MMDR	15
MXRDR	16
MXRAR	17
IR△B2	19
MQ	1A
IR△X2	1B
IR△OP	1C
IR△R1	1D
UFRDR	1E
LFRDR	1F
IR△D2	20
PSW	21
AC	22

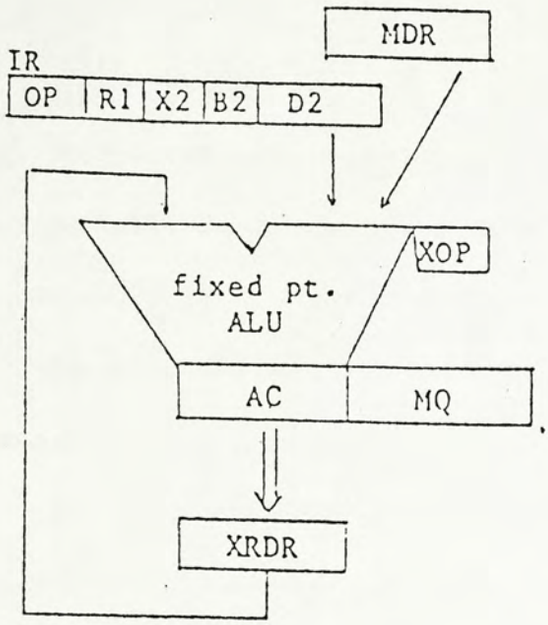
Fig. 6.5 Addresses of registers of the Chen's machine

The corresponding control words for the ALUs, memory are also shown in Fig.6.5 as follows.



*for single precision arithm.
*in hexadecimal representation

Fig.6.6a Operating the floating point ALU



*in hexadecimal representation

* control word for the fixed-point ALU:

operand--XRDR,MDR
- 5A : add
- 5B : subtract(XRDR-MDR)
- 5C : multiply
- 5D : divide(XRDR÷MDR)
- 54 : AND
- 56 : OR
- 5E : NAND
- 5F : NOR

operand--XRDR,IR/D2
- 50 : add

operand--XRDR
- 52 : decrement by 1
- 53 : increment by 1
- 89 : left shift by 1 bit; 0 enters LSB
- 88 : right shift by 1 bit; 0 enters MSB
- 8E : rotate leftward by 1 bit

Fig.6.6bControl words for the fixed-point ALU.

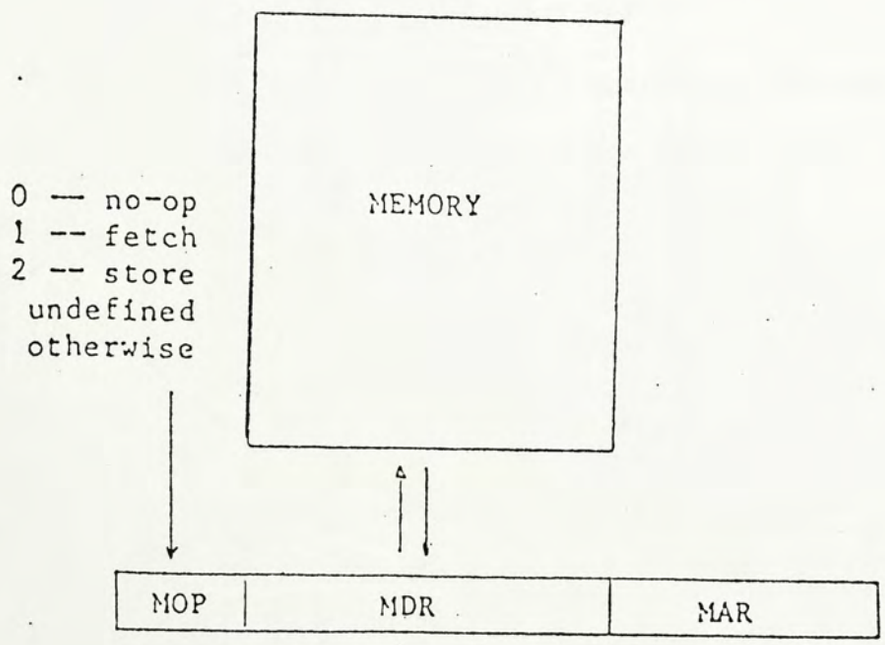


Fig. 6.6cMemory fetch and store

6.2 Microinstruction Description in MIMOL

As will be shown, the Chen's MI can completely be described by MIMOL. The description itself also presents a clear picture of the microinstruction in a hierarchical fashion that itself is already self-explanatory.

In the following sections, different segments of the MI description on the machine's resources, intra-field details, inter-field relations, machine defined testable conditions and implied operations respectively are presented.

One special remark, however, is needed to make. It is about the field enumeration of the "source" field (lines 136-169). Readers may think that it is too troublesome to type in these lines as they are repeating those of the "sink" field (lines 135-170). However, this is only half-right. For in VS/APL, those lines for the "sink" field (i.e. lines 135-170) can easily be duplicated by first storing them as a variable. Then insert this variable (using VS/APL "concatenate" operators) between the two lines delimiting the "source" field enumeration (i.e. lines 135 and 170).

6.2.1 Resource Declaration

```
 1  RESDEL
 2
 3    CMT 'CONTROL STORE; MAIN MEMORY'
 4    'CS' OF MATRIX 1024,24 POINTER'MMAR'
 5    'MM' OF MATRIX 5120,32 POINTER'MAR'
 6
 7    CMT 'CONTROL UNIT REGISTERS'
 8    'MIR; MMDR' OF SEQ 24
 9    'MPC; MMAR; MXRDR' OF SEQ 16
10    'MXRAR; MXROP; MMOP' OF SEQ 8
11
12    CMT 'MICROINDEX REGISTERS'
13    'MXR' OF MATRIX 4,16 POINTER'MXRAR'
14
15    CMR 'CONSTANTS'
16    'MXR[0;]' EQU 0
17    'MXR[1;1]' EQU 1
18
19    CMT 'OPERATION UNIT REGISTERS'
20   'FRDR' OF SEQ 64
21    'IR;MDR;UFRDR;LFRDR;XRDR' OF SEQ 32
22
23    CMT 'STACK REGISTER'
24    'STK' OF SEQ 40
25    'MAR;PC' OF SEQ 24
26
27    'MOP;INDC;MASK;FRAR' OF SEQ 8
28    'FROP;XRAR;XROP;AOP;XOP' OF SEQ 8
29
30    CMT 'ADDRESSABLE SUBREGISTERS'
31
32    'IRΔOP;IRΔR1' OF SEQ 8
33
34    'IRΔX2;IRΔB2' OF SEQ 4
35
36    'IRΔD2' OF SEQ 12
37
38    CMT 'WORK REGISTER FILES'
39
40    'XR' OF MATRIX 16,32 POINTER'XRAR'
41    'FR' OF MATRIX 7,64 POINTER'FRAR'
42
43    CMT 'CONSTANT'
44    'XR[0;]' EQU 0
45
46  ENDRES
```

6.2.2 Intra-Field Declaration

(i) Field Identification

```
47
48  FIELDS 'CS'
49
50    FMT 'F0' WITH MI[0,8] EQU 0,0
51
52      SECTION 'S1'
53        'SINK' ATX 1,6
54        'A' AT 7
55      ENDSECTION
56      'DATA' ATX 9,23
57    ENDFMT
58    FMT 'F1' WITH MI[0,8] EQU 0,1
59      INCL S1
60      'SOURCE' ATX 9,23
61    ENDFMT
62
63    FMT 'F2' WITH MI[0] EQU 1
64      'STACKOP' AT 1
65     'JUMPTEST' AT 2
66      'TESTBITOP' AT 3,4
67      'REGNAME' ATX 5,8
68      'BITOP' ATX 9,13
69      'ADDREG' AT 14,15
70      'DISP' ATX 16,23
71    ENDFMT
72
73  ENDFI
```

(ii) Field Enumeration

```
74
75  FDETAILS
76
77    OPTFIELD 'A' DEF 0
78      'NOP' EQU 0
79      'INC1' EQU 1
80    ENDOPT
81
82    OPTFIELD 'STACKOP' DEF 0
83      'NOP' EQU 0
84      'PUSH' EQU 1
85    ENDOPT
86
87    OPTFIELD 'JUMPTEST'
88      'IF0' EQU 0
89      'IF1' EQU 1
90    ENDOPT
```

(ii) Field Enumeration (cont'd)

```
 91
 92    OPTFIELD 'TESTBITOP' DEF 2
 93      'RESET' EQU 0
 94      'NEG' EQU 1
 95      'SET' EQU 3
 96    ENDOPT
 97
 98    OPNFIELD 'SINK' DEF 0
 99    'AOP' EQU 0
100    'FROP' EQU 1
101    'XOP' EQU 2
102    'XROP' EQU 3
103    'MOP' EQU 4
104    'STACK' EQU 5
105    'MXROP' EQU 6
106    'MMOP' EQU 7
107    'IR' EQU 8
108    'INDC' EQU 9
109    'MASK' EQU 10
110    'FRDR' EQU 11
111    'XRDR' EQU 12
112    'MDR' EQU 13
113    'PC' EQU 14
114    'MAR' EQU 15
115    'FRAR' EQU 16
116    'XRAR' EQU 17
117    'MIR' EQU 18
118    'MPC' EQU 19
119    'MMAR' EQU 20
120    'MMDR' EQU 21
121    'MXRDR' EQU 22
122    'MXRAR' EQU 23
123    'IRΔB2' EQU 25
124    'MQ' EQU 26
125    'IRΔX2' EQU 27
126    'IRΔOP' EQU 28
127    'IRΔR1' EQU 29
128    'UFRDR' EQU 30
129    'LFRDR' EQU 31
130    'IRΔD2' EQU 32
131    'PSW' EQU 33
132    'AC' EQU 34
133    ENDOPN
134
135    OPNFIELD 'SOURCE' DEF 0
136    'AOP' EQU 0
137    'FROP' EQU 1
138    'XOP' EQU 2
139    'XROP' EQU 3
140    'MOP' EQU 4
141    'STACK' EQU 5
142    'MXROP' EQU 6
```

(ii) Field Enumeration (cont'd)

```
143     'MMOP' EQU 7
144     'IR' EQU 8
145     'INDC' EQU 9
146     'MASK' EQU 10
147     'FRDR' EQU 11
148     'XRDR' EQU 12
149     'MDR' EQU 13
150     'PC' EQU 14
151     'MAR' EQU 15
152     'FRAR' EQU 16
153     'XRAR' EQU 17
154     'MIR' EQU 18
155     'MPC' EQU 19
156     'MMAR' EQU 20
157     'MMDR' EQU 21
158     'MXRDR' EQU 22
159     'MXRAR' EQU 23
160     'IRΔB2' EQU 25
161     'MQ' EQU 26
162     'IRΔX2' EQU 27
163     'IRΔOP' EQU 28
164     'IRΔR1' EQU 29
165     'UFRDR' EQU 30
166     'LFRDR' EQU 31
167     'IRΔD2' EQU 32
168     'PSW' EQU 33
169     'AC' EQU 34
170     ENDOPN
171
172     OPNFIELD 'REGNAME' DEF 0
173     'AOP' EQU 0
174     'FROP' EQU 1
175     'XOP' EQU 2
176     'XROP' EQU 3
177     'MOP' EQU 4
178     'STACK' EQU 5
179     'MXROP' EQU 6
180     'MMOP' EQU 7
181     'IR' EQU 8
182     'INDC' EQU 9
183     'MASK' EQU 10
184     'FRDR' EQU 11
185     'XRDR' EQU 12
186     'MDR' EQU 13
187     'PC' EQU 14
188     'MAR' EQU 15
189     ENDOPN
```

(ii) Field Enumeration (cont'd)

```
190
191     OPNFIELD 'ADDREG' DEF 0
192       'MXR[0;]' EQU 0
193       'MXR[1;]' EQU 1
194       'MXR[2;]' EQU 2
195       'MXR[3;]' EQU 3
196     ENDOPN
197
198     EMFIELD 'DATA' DEF 0
199       RANGE 0,32767
200     ENDEM
201
202     EMFIELD 'BITPO' DEF 0
203       RANGE 0,31
204     ENDEM
205
206   ENDFD
```

6.2.3 Inter-Field Declaration

```
207
208   INFIELDS 'CS'
209     SOURCE TO SINK
210     EMIT TO SINK
211     (ADDONE SOURCE) TO SINK
212     (ADDONE EMIT) TO SINK
213     STACKOP
214     TESTBITOP
215     JUMPTEST REGNAME,BITPO
216     GOTO ADDREG,DISP
217
218   ENDINF
```

6.2.4 Machine Defined Testable Condition Declaration

```
219
220  CONDITION
221
222    EQUOP 'IF1 REGNAME,BITPO'
223    EQUOP 'IF0 REGNAME,BITPO'
224
225  ENDCON
```

6.2.5 Implied Operation Declaration

```
226
227  IPOP 'JUMPTEST REGNAME,BITPO'
228
229    IMPLY MUST 'GOTO ADDREG,DISP'
230
231  ENDIP
```

6.3 AMPL for Chen's Machine

6.3.1 Overview

As described earlier, AMPL itself is just a schema language. In order to have "decent" microprogramming of a particular machine, an "instantiation" from AMPL to obtain the necessary microprogramming language is required. As AMPL basically consists of two major types of language construct: the declaration and the execution constructs, then the general structure of a program written by the microprogramming language developed from AMPL for the Chen's machine (hereinafter called AMPL(C)) is also bound by these constructs and shapes as below:

```
MPG '<program name>'
  DECLARATION
     ...            /pseudovariables declarations; synonyms
     ...             for renaming, constant declarations and
                     macro definitions
  ENDEC
  BEGIN
     ...            /executable statements running in the
                     program
     ...
     ...
  PROC '<procedure name>'
    DECL            /optional synonym and literal constants
     ...             declarations apply to this procedure
     ...             only;
```

(cont'd)

```
        ENDECL
        BEGIN
          ...              /executable statements contained in the
          ...               procedure
          ...
        END
      ENDPROC
      ...                  /executable statements in the program
      ...
      ...
  END
ENDMPG
```

Further illustration of these constructs will be described later.

The declaration block consists of data types and data object constructs for declaring pseudo-variables to be used globally throughout the program. Pseudo-variables are the type of variables having no corresponding with the host machine resources. Yet, there introduction may, at some times, brings programming conveniences. As the data objects used by the program are machine specific and already predefined in the microinstruction description, the programmer basically need only to declare the pseudo-variables, literal constants, and performs renaming of data objects. Macros used in the program must also be declared in the declaration block. While for the execution block, it consists of executable statements and one or more procedures, which in turn consist of one or more executable statements.

6.3.2 Executable Statements

The executable statements in AMPL(C) basically retain the shapes as those described in ,section 3.2. Essentially, the executable statements of AMPL(C) consists of:

(1) Assignment statements of the form

<source> TO <sink> (VI.1)

where <sink> is a register variable and <source> may either be a register variable or constant; or even a parenthesized expression taking the form of a function statement as described below.

(2) Function statement of the form

<operation identifier> {<operand> {,<operand>}} (VI.2)

where <operation identifier> is constrained by the Chen's machine allowable operation keywords declared in section 6.2; <operand> here may refer either to register variable or constant. Please note that although the Chen's microarchitecture only provides a limited set of operation keywords, a large number of operators (in keyword form) described in section 3.2 in fact are retained as most of them can be declared as macros in the AMPL(C) program (see section 6.4 and Appendix A).

(3) Branching statements of the form

GOTO <address> (VI.3)

or ACT '<procedure identifier>' (VI.4)

where <address> may either be a label or a literal constant or a register variable selected from MXR[0;] or MXR[1;] or MXR[2;] or MXR[3;]. In case the <address> referred is a label address, quotes are required as for the ACT command. ACT (means activate) in here is a program reserved word. It is essentially a non-return call to a procedure as specified by the identifier following it.

(4) Conditional statement of the form

IF '<program condition or test condition>' ... ENDIF (VI.5)

where the <test condition> is as defined in section 6.2.4. In the Chen's machine, this <test condition> is the testing of a bit value of a register in the machine organization. This is very powerful as a large number of machine registers (from address 00 to 0F in hex), and wide range of bit positions (from position 0 to 31) can be tested.

For the <program condition>, it is referring to the type of condition relating to the testing of a certain virtual objects like literal constant or pseudo-variable.

(5) Compound statement of the form

COBEGIN S1;S2;...;Sn ENDCO (VI.6)

where Si may be one of the statements VI.1-4 depending on the context of the microinstruction one would like to compose. This is also the only statement that requires the programmer's knowledge of the control store organization. This is just as expected (and required) for this statement is used by the programmer at the time he wants to optimize his microprogram at the source level.

(6) Repetition statement of the form

REPEAT <statement VI.3 or VI.4> UNTIL '<test condition>
or
<program condition>'

(VI.8)

WHILE '<test condition>
or
<program condition>' DO <statement VI.3 or VI.4>

(VI.9)

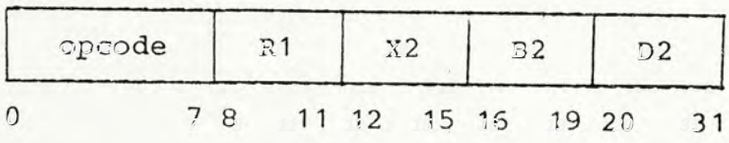
where REPEAT, UNTIL; WHILE, DO are program reserved words. The semantics of the repetition statement follows those described in the conditional statement (VI.5). In case where the <program condition> (see VI.5) is used rather than the <test condition>, there is no restriction to the type of statements following the REPEAT and DO keywords (see Appendix A for the emulator listing). Under such a case, normal expansion like macro expansion results.

6.4 A Sample-- An Emulator

6.4.1 The Emulated Macroarchitecture

A sample microprogram, which basically is an emulator - a set of microprograms that implement the control of the Chen's machine-- resides in the control store, is written. The emulated macroarchitecture basically resembles that of the IBM S/360 and is an abstraction of the one presented in [3]. Fig.6.6 shows the emulated macroarchitecture with the action column machine resources referring to those described in the Chen's machine organization (Fig.6.3).

Macroinstruction format:



Macroinstruction:

Opcode	Mnemonic	Action	
70	STE	Store floating	FR(R1)->M(EA)
50	ST	Store	XR(R1)->M(EA)
51	STDX	Store double XR's	XR(R1,R1+1)-> M(EA,EA+1)
58	LD	Load	M(EA)->XR(R1)
7A	AE	add floating	FR(R1)+M(EA)-> FR(R1)
7B	SE	subtract floating	FR(R1)-M(EA)-> FR(R1)
7C	ME	multiply floating	FR(R1)xM(EA)-> FR(R1)

Fig. 6.6 The Emulated Macroarchitecture (to be cont'd.)

7D	DE	divide floating	FR(R1)÷M(EA)->FR(R1)
5C	M	multiply	XR(R1)xM(EA)->XR(R1)
6B	SI	subtract immediate	XR(R1)-EA->XR(R1)
5E	NA	NAND	XR(R1)∧M(EA)->XR(R1)
6F	NOI	NOR immediate	XR(R1)∨EA->XR(R1)
88	SRL	shift right logical	XR(R1)÷2->XR(R1); 0->MSB
8E	ROL	rotate leftward	rotate XR(R1) leftward EA times
43	CAL	call subroutine	PSW->STACK;EA->PC
44	RETN	return from subroutine	STACK->PSW
80	SMASK	set MASK	MASK D2->MASK
46	BCNT	branch on count	XR(R1)-1->XR(R1)
82	SINDC	set INDC	INDC D2->INDC
48[*]	BSTKF	branch on stack full	if STKF=1, EA->PC
84	STMSK	store MASK	MASK->XR(R1)
4C[*]	BARN	branch on negative	if ARN=1, EA->PC
85	LDMSK	load MASK	R1->MASK
86	STIND	store INDC	INDC->XR(R1)
8B	NOOP	no-operation	PC+1->PC

Fig.6.6 The Emulated Macroarchitecture (cont'd)

Note: [*] If R1=0: test bit is reset to zero,
1: negated,
2: unchanged,
3: set to one.

EA (effective address) in the above statememts is calculated by EA=XR(X2)+XR(B2)+D2

6.4.2 Design of the Emulator

A flowchart of the emulator is shown in Fig.6.7. For the execution of each of the macro-instruction, the emulator first calculates the effective address for the second operand, and then decode the opcode. Decoding of the opcode is made with the aid of a jump table. Each entry of the jump table is a branch microinstruction. Execution of one of the branch microinstruction leads to the beginning of a function routine which is a microprogram performing the function of one or more of the machine instructions. The address of each entry of the jump table coincides with the opcode of one of the macro-instructions.

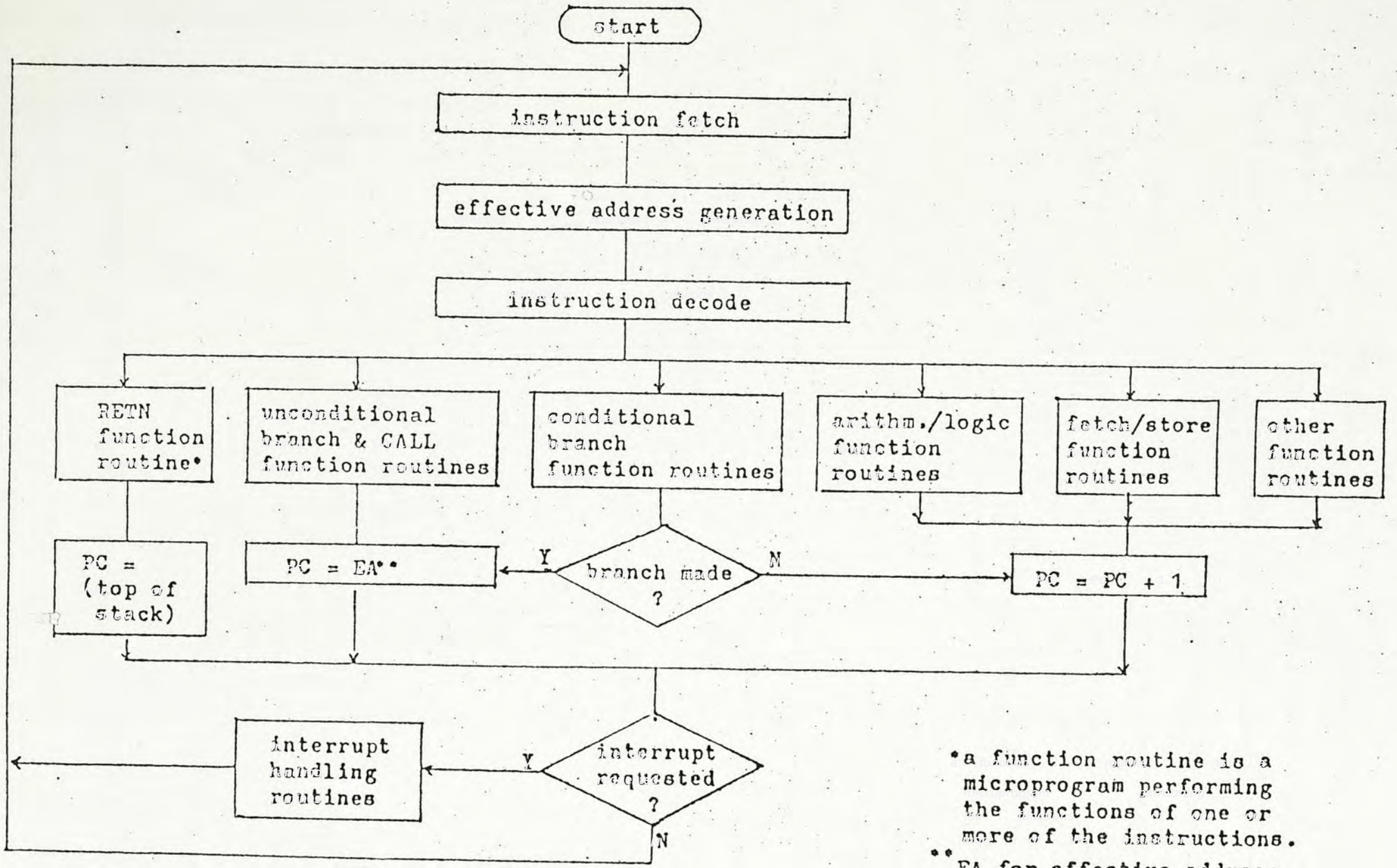


Fig. 6.7 A flowchart for the emulator.

Fig.6.8 shows an example of how the opcode of the machine instruction ST (store) is decoded. First the emulator moves the opcode into MPC. The next microinstruction fetched from the control store is a branch microinstruction from location 50 (in hex). Execution of this branch microinstruction then directs the microprogram flow to the beginning of the STore function routine.

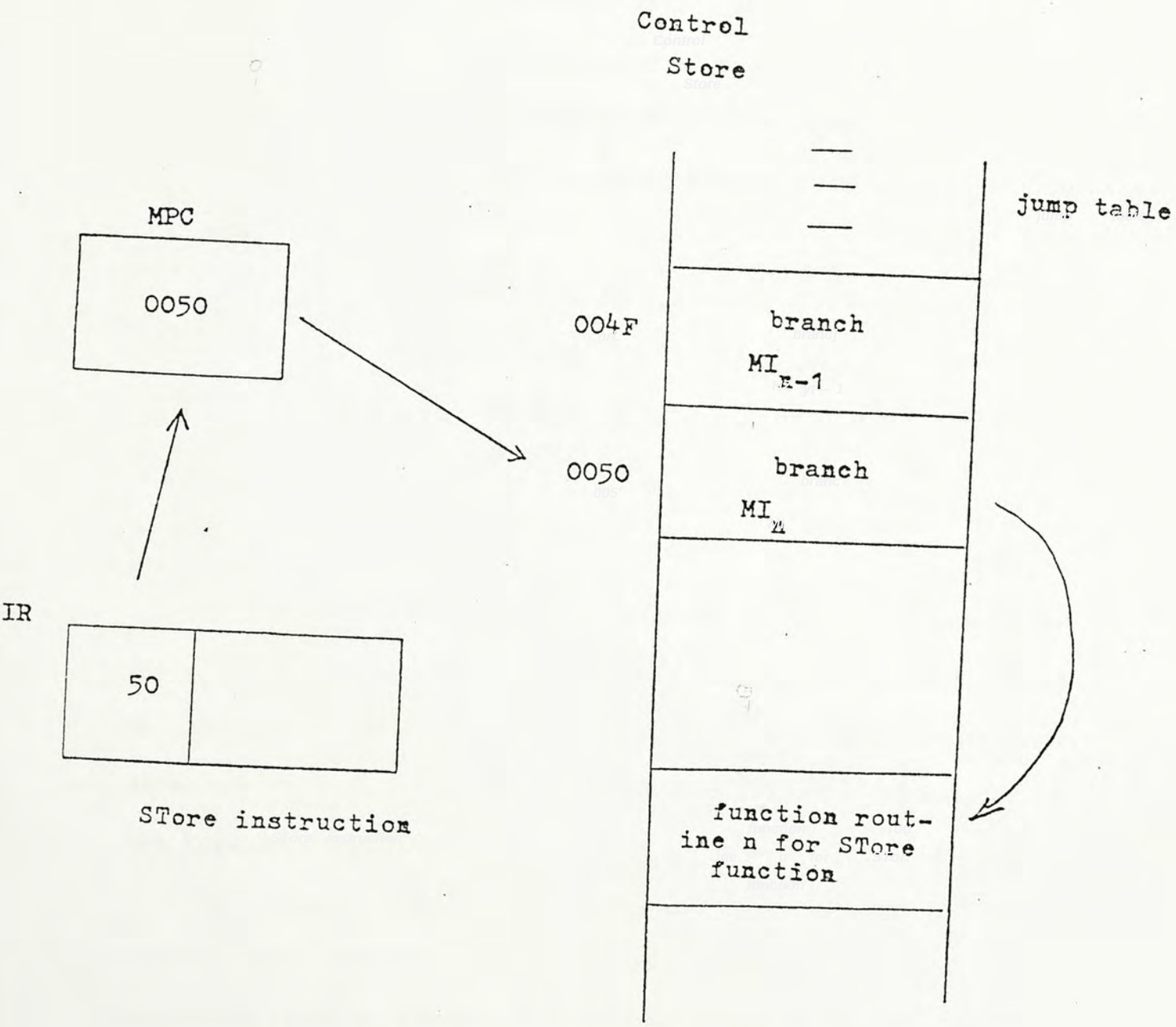


Fig.6.8 An example of decoding using the jump table

6.4.3 The Emulator Program

A full listing of the emulator written in AMPL(C) for the macroarchitecture depicted in Fig. 6.6 is shown in Appendix A. Referring to the Appendix, the first feature we can readily observe in the emulator written in AMPL(C) is the abstraction of function using macros. In fact, these macro functions are declared after we have analysed more than two hundred and fifty lines of microcode statements [3]; those frequently used microcode segments are then declared as macros. And these macros are all declared without the need to know about the machine control store organization. As shown by the LOAD macro functions below:

```
MACRO 'LOAD A,B'
B TO MAR
1 TO MOP
MDR TO A
ENDMACRO
```

The declaration of a macro function is fairly simple. We need only start the declaration using the keyword MACRO followed by the macro-heading; then write all the executable statements necessary for the macro function in the macro body and bind it by the keyword ENDMACRO.

As AMPL(C) itself is basically a keyword type language, writing and reading the emulator is highly facilitated by the provided keyword mnemonics. As illustrated in the emulator listing, one only need the machine organization chart and

declared operation keywords (section 6.2) at hand and can readily write his microprogram in AMPL(C) without any need to know the control store organization.

Another feature that is worth mentioning is the structure of the emulator. As shown, the emulator is well structured and readable. Indentation and blank lines can also be allowed to be introduced in the source program to improve its readability. Although the introduction of the program reserved words (e.g. BEGIN, END, etc.) for better program structuring has increased the number of program statements (roughly 10%), they virtually require no programmer's effort. Even without the code production phase of the AMPS, it is believed that the emulator can be hand translated to microcode without much difficulties.

Below is a brief analysis on the emulator written. Referring to the emulator listing in Appendix A, the first line is the program heading indicating that the program name is "EMULATOR". Lines 4-15 are for synonyms declaration. These declared synonyms will then be valid globally throughout the entire program (i.e. the emulator). Lines 17-188 are for the macros declaration. The following is a list illustrating the meanings of these declared macros (machine resources are abbreviated as in the resource declaration, section 6.2.1)

line no. in the emulator listing	macro-heading	meaning
17	LOAD A,B	A<--MM(B)
23	STOR A,B	MM(B)<--A
29	LOAD CS A,B	A<--CS(B)
35	STOR CS A,B	CS(S)<--A
41	LOAD MXR N,B	B<--MXR(N)
47	STOR MXR N,B	MXR(N)<--B
53	LOAD XR N,B	B<--XR(N)
59	STOR XR N,B	XR(N)<--B
65	LOAD FR N,B	B<--FR(N)
71	STOR FR N,B	FR(N)<--B
77	INFET	/ instruction fetch
83	GENADD	/ generate address
90	FDIV INDEX,MDR	/ floating divide FR(INDEX)-MDR
97	FADD INDEX,MDR	/ floating add FR(INDEX)+MDR
104	FSUB INDEX,MDR	/ floating subtract FR(index)-MDR
111	FMUL INDEX,MDR	/ floating multiply
117	BADD XRDR,MDR	/ binary add XRDR+MDR
122	BSUB XRDR,MDR	/ binary subtract XRDR-MDR
127	BMUL XRDR,MDR	/ binary multiply XRDR MDR
132	BDIV XRDR,MDR	/ binary divide XRDR-MDR

```
137     AND XRDR,MDR        XRDR MDR
142     OR XRDR,MDR         XRDR MDR
147     NAND XRDR,MDR       XRDR MDR
152     NOR XRDR,MDR        XRDR MDR
157     ADD XRDR,IR D2      XRDR IR D2
162     DEC XRDR            XRDR-1
167     INC XRDR            XRDR+1
172     SHAL XRDR,N         / XRDR arithmetic
                              left shift by N
                              places
178     SHAR XRDR,N         / XRDR arithmetic
                              right shift by N
                              places
184     SHLL XRDR,N         / XRDR logical left
                              shift by N places
```

It can be observed that most of the operation keywords in section 3.2.3 are retained as macros in the emulator written.

After the declaration of the macros, the execution blocks begins at line 192. In fact, the execution block consists of many series of procedures interlaced with basic microstatements. Many procedures actually take the names of the macroarchitecture opcode which they emulate as their procedure names. The procedure "ST" is one such example. Readers may note that there are synonyms declared in many procedures, for examples in lines 211 and 222. Essentially, they are the synonym declarations for "EA" (effective address). Since the scope of synonyms declared within a procedure is confined within the procedure, "EA" is not valid outside the procedure which has declared it. As a result, the "MAR" can still be available for some other procedures.

Another point needs mentioning is about the machine defined testable conditions statements appeared in some procedures like "BSTKF" and "BARN". As these machine defined testable conditions have the branching operation as the implied operation, thus there were always the "GOTO" or "ACT" (basically the same nature as "GOTO") statements following them. As a result, an implied parallelism exists between the implying and implied operations at the source level. However, the different "condition" statements appeared in these procedures did not suggest that they are to be executed in a concurrent manner.

One final point about the emulator is the jump table. It is placed at the end of the emulator listing (starts at line 564 and ends at line 584). Since only those procedures/routines interpreting the macroarchitecture opcodes are to be accessed via this jump table, other procedures in the emulator will therefore not be activated by this jump table.

In the following section, the microcode output for the emulator is discussed.

6.5 The Microcode Output

The emulator written in AMPL(C) is successfully translated into microcode. Interested readers may refer to Appendix B for the full listing.

Referring to the microcode listing, it starts at address 0145 (decimal) because the address 0-144 are reserved for the jump table (Fig.6.8). As shown, the generated microcode for the emulator is highly compacted though no microcode optimizer or compacter is incorporated into the AMPS. (Only a hundred and eighty-eight lines of microcode are generated for the emulator which in turn takes more than five hundred lines at the source level)! This can be explained as follows. Referring to Fig. 6.1, there are basically two types of available microinstructions for the Chen's MI, namely the register transfer and branch microinstructions. In fact, these two types of microinstructions can be directly mapped with the basic executable statements of the AMPL(C). As a result, the microprogram written in AMPL(C) basic executable statements will therefore be self-optimized at the source level. Thus we can have a highly compacted microcode output which we believe is compatible with a hand-coded one, but with far less effort.

Chapter 7. Conclusion

7.1 Discussion

Although the theme of this thesis is on the design and construction of an AMPS, it basically consists of the works on

(a) the proposal of an approach for automatic microcode generation;

(b) the design of a higher level microprogramming language, namely AMPL;

(c) the design of a microinstruction description language, namely MIMOL;

(d) the implementation of an AMPS prototype system for the Chen's machine.

For this discussion section, it is also presented as accorded to the above four main topics.

(a) On the proposed approach:

One most significant point incurred after the implementation of the AMPS about the proposed approach for automatic microcode generation is that it does work. Although more vigorous tests and possibly modification(s) may be required, yet we can still expect that the approach can apply for a class of machines have the type of microarchitecture similar to that of the Chen's machine. Moreover, though it is just a modest approach yet it does provide a "seemed-possible" attempt

to tackle the long posted problem of retargetability of microcode generation system.

(b) On AMPL:

As previously mentioned, AMPL is an S* [1] influenced language. However, S* cannot be directly used here as our microprogramming language nor can it be treated as equivalence to AMPL due to the following reasons.

First, the proposed approach requires the operators for the basic statements of the microprogramming language to be of the keyword type. However, S* employs a lot of operators [1] which are therefore contrary to the just mentioned "keyword type" notion. Besides, these operators also appear quite peculiar of its own (for example $\hat{\ }$, $\square$, etc.) that reduce the degree of portability of the language and make it not in concordance with our primary goal set. Second, we feel that at our current experimental stage for the proposed approach for automatic microcode generation, it is not necessary for us to include the abundance of S* operators and statements for depicting microparallelism. In fact, these opertors and statements also require one to have ultimate knowledge about the machine control store organization, which is really not we want. Though S* may suggest that AMPL, in its future course of development, may need to include those operation keywords and statements for

handling more sophisticated microparallelism; however as far as our current implementation is concerned they are not required. Third, some important language constructs like macro facilities, non-return call of routine commands, etc. which are deemed important in "real" microprogramming context are not included in S* [1], [16]. In addition, the basic operators set which is supposed to be referenced as the frame for latter "instantiated" versions of S* was also not very clearly reported in the original papers [1] and [16].

(c) On MIMOL

Although MIMOL posseses the meritorious point that it can allow one to define Chen's machine microinstruction in a completely machine independent manner, readers may still think that using MIMOL may be a bit tedious for microinstruction description. This may largely in part due to the MIMOL MI field enumeration phase. For in this phase, every resource or operation keyword for a particular field has to be enumerated. Although the MIMOL microinstruction description is only a once-off job to the users, the field enumeration phase may grow very tedious as the number of machine resources (accordingly the number of operation keywords also) grow. These all may suggest that MIMOL should have higher abstraction capability for intra-field description.

In addition, the MIMOL can also be further developed to allow itself to describe more sophisticated

microprogram control flow activities, e.g. microtraps, micro-subroutine calls, etc.

(d) On the implementation:

The implementation does improve the "effectiveness" of the Chen's machine by bringing to it a lot of programming conveniences. Moreover, experiences are also gained on the various aspects concerning automatic microcode generation after the implementation. They are also discussed as below.

(1) In order to have proper automatic microcode generation, a discipline is mandatory. This discipline, bases on our experience gained from the implementation should be uniform and applied coherently down from the machine drawings to the writing of the microprogram. For example, resources names depicted in the organization diagram and the MI diagram must be consistent. Usually, the MI fields in the original diagram (from the MI original designer) may not be depicted as according to our designated field types. Therefore the MI fields should be clearly identified of their types before the MI is described using the MI description language. For our case under the proposed approach, it is required that MI fields has to be classified as one of the thr typ , r ly the operation

select, operand select and the emit field. For example the two fields 'regname' and 'bitpo' of the Chen's machine MI are originally combined and named 'testreg' in the original diagram.

(2) With regard to our microprogramming experience with the AMPL, we find that the "structure" of a microprogram is indeed very important. In our case for example the high degree of readability of the emulator written really helped us a lot during our "walkthrough" process. This point is also true in our case for the microinstruction description.

(3) Another point that's worth mentioning is about the using of VS/APL. As well understood, VS/APL is working in an interactive mode. However, the AMPL program requires its translator to work in a "compiler" like fashion. Therefore, this demands certain unusual applications of VS/APL. In fact the emulator is firstly constructed as a character matrix in VS/APL so as to allow indentation and blank lines to be included to the emulator "program" to improve its readability. As the VS/APL editor is only a line editor, editing job would be much more easier if APL2 [16] could be used. APL2 is basically an advanced version of APL which has 2-dimensional editing capability. During execution, the character

matrix (i.e. the emulator) is actually converted into a VS/APL program using the ⎕FX function. During the declaration part of the program, a declaration mode flag is set such that all the VS/APL bit-setting functions, after seeing the set flag, skip. Thus basically there is no execution problem for the declaration block of the AMPL program. However, problems do come for the execution block of the emulator for the GOTO, ACT and REPEAT...UNTIL statements. For the ACT and GOTO bit-setting functions. It is really difficult for them to look ahead in VS/APL and predict a correct address for the procedure they are "superficially" calling and set this address to the MI word. However, this problem can be solved by first storing up the address of this GOTO or ACT statement in the presumed object code matrix and later correct the corresponding row bit pattern at the end of the emulator. For the REPEAT..UNTIL statement, VS/APL actually has no direct support of such kind of statement. Yet we can get around the problem by using the ⎕LC system function as follows:

```
∇ Z<--UNTIL '<condition>'
[1]  Z<--((⎕LC[1]+1), ⎕LC[1]-1)[(<condition>=true)+1]
∇
```

Since the REPEAT statement is always at the

(⎕LC[1]-1) position with respect to the UNTIL statement, therefore this line number is referenced in the UNTIL function. In fact, the REPEAT..UNTIL statement is actually implemented in VS/APL as follows

```
REPEAT '<action'
-->UNTIL '<condition>'
```

Thus this solution to the REPEAT..UNTIL statement is therefore basically different from the APLGOL [17] as APL syntax can be preserved in our case.

In addition, certains sub-topics are also derived after the implementation that worth separate discussion. They are

i) the Chen's machine:

Referring to Fig.6.1 the Chen's machine MI and Fig.6.3 the Chen's machine organization, it is believed that the machine has been "elegantly" designed to illustrate the basic concepts of microarchitecture. As illustrated in these figures, nearly all kinds of basic machine operations can be found displayed by the Chen's machine MI, for example the push of machine stack, the uncondition/condition branch with operation on the test bit, the selectable operation: setting, resetting or negating of a bit on some of the machine registers under a particular test condition, etc.

Yet, the Chen's machine can still be further

improved. The first is about the MI. The MI can be lengthened so that two or more register transfer operations can be initiated by the same MI. The emulation process will then be speeded up for previously long register transfer operation sequences are shortened (assuming there are no resource and data conflicts). Moreover, the 'disp' field addressing space can also be increased (the original space is only 255). In addition, adding bits in the MI gives a chance to include the previously left over micro-stack pop operation.

Second, it is on the Chen's machine hardware. Bus system(s) can be introduced so that hardware concurrency can be realized. At the mean time, this also allow one the chance to alter the system "internal configuration" (i.e. by grouping of certain registers for a particular bus) for hardware "tuning" purpose so that the best performance for a particular application can be achieved. From system architecture education viewpoint this also carries high pedagogic values. Moreover, a more advanced system clock phasing circuit can be introduced into the original hardware. This will further allow the Chen's machine to have the chance to exercise pipelining operations. Besides in bringing higher system performance, it meanwhile gives the machine the capability and flexibility (if the system "tuning" is allowed) to closer resemble some polyphase machines. Under such circumstances, the effectiveness of

various higher level microprogramming languages can be tested and evaluated within one single machine thus incurs higher value, from both educational and practical viewpoints.

ii) future research:

The implemented AMPS does successfully generate microcode and is verified correct (by hand). However, several entities can be added to the currently implemented AMPS to make it a more complete microcode development tool. The first is a microcode optimizer or compactor and the second is a microcode verifier/simulator. And due to the uniqueness of the Chen's machine MI, the later entity is comparatively more needy in our situation. In fact a microcode verifier has virtually been built by Lai [3] two years ago. Basically what he did was the construction of a "hardware-simulated" version of the Chen's machine with the emulator (as presented in section 6.4) installed in it. Panel displays and control switches were also built so that the many registers' content of the Chen's machine could be shown and the microcode kept in the control store could be invoked, executed and verified using the machine built. However the emulator was done by direct microcoding. Hence a lot of Lai's work on the construction of the emulator could be saved by the currently implemented AMPS. Nonetheless, the two pieces of work, namely the software and the hardware versions of the Chen's are complementing each other rather than

overlapping in their functions.

7.2 Summary

In this thesis, an approach towards automatic microcode generation via the design and construction of an automatic microprogramming system is presented. Under such an approach, rather than constructing a translator for the microprogramming language, only an once-off description of the microarchitecture is required. Bases on this description, the AMPS then generates the appropriate functions to translate source higher level microprograms to microcode. Two lanugages, namely AMPL (A Microprogramming Language) and MIMOL (Microinstruction Modelling Language) are also designed for the AMPS.

Before the detailed constructs for the two languages are presented, the AMPS basic organization and the proposed approach are discussed and illustrated with example. The many difficulties and various considerations for higher level microprogramming, namely

(i) the resource binding problem,

(ii) machine specificity and variability of microarchitectures,

(iii) the allowable degree of abstraction,

(iv) the degree of machine independence and higher-levelness of the language used, and

(v) user microprogrammability

are also pinpointed and discussed.

AMPL is a higher level microprogramming language that can be applied to work for various machines. Its generality in application to different microarchitectures is achieved by the

method of instantiation [1]. AMPL is instantiated into a microprogramming language AMPL(M) by declaring instances of the idiosyncratic details of the machine M into AMPL. In fact, instantiated versions of AMPL only differ in their elementary statements and implied operations.

The language AMPL is also easy to be grasped for it is basically a mnemonic type language. User microprogrammability can be achieved for the keyword commands of AMPL is well self-explanatory. In addition, AMPL also retains the flavours of conventional high level languages such as structured control constructs so as to facilitate the writing of structured and readable microprograms. Thus reliable microprograms can be written. Besides, a significant degree of function abstraction is also allowed by the declaration of macros using the AMPL facilities.

MIMOL, on the other hand, is a language for the description of the various aspects of a microarchitecture, namely

(i) the machine organization hardware resources ,

(ii) the intra-field details of the microinstructions involved,

(iii) the inter-field relations incurred by the nature of different fields of a microinstruction,

(iv) the machine defined testable conditions,

(v) the implied operations between operations/operands of different fields of a microinstruction.

In addition to allowing users to define the semantics of microinstructions, the formalism adopted in MIMOL also allows

users to describe microarchitectures in a completely machine-independent manner. Thus, one of the primary aim of designing AMPS, namely, retargetability, is accordingly satisfied.

Certain fundamental goals have been attained by the designed AMPS, namely,

(a) Retargetability, i.e. AMPS is applicable to different machines. Here retargetability is applying for the class of machines having the type of microarchitecture similar to that of the Chen's machine;

(b) User microprogrammability, i.e. AMPS allows users to practise higher level microprogramming with the easy to understand keyword type language, namely AMPL;

(c) Transportability, i.e. AMPS can be implemented on different machines without much difficulties;

(d) Integrity in the development of microcode. This is achieved as both AMPL and MIMOL are applied in a coherent fashion for microcode development in AMPS;

(e) Automatic generation of microcode, i.e. no dedicated translator for the microprogramming language of any particular machine is required to construct for the generation of microcode.

The feasibility of the AMPS is also demonstrated in this thesis through the construction of a prototype system for the Chen's machine [2]. Microcode is also generated for an AMPL microprogram emulating an architecture resembling that of the IBM S/360. Together with the hardware version of the Chen's machine

[3], the currently implementated system also carries high pedagogic values.

References

[1] Dasgupta, S, "Some Aspects of High-Level Microprogramming", Computing Surveys, V.12, No.3, Sept. 1980 pp.295-323.

[2] Chen, T. C., Lectures notes on microprogramming in the Chinese University of Hong Kong, 1982.

[3] Lai, K. W., " A Visible CPU Via Elementary Microinstructions ", M. Phil. Thesis, Chinese University of Hong Kong, 1983.

[4] Wilkes, M. V., "The Use of a Writable Control Memory In a Multi-programming Environment ", ACM 6th Annual Workshop on Microprogramming (Preprints), 1972 pp.62-65.

[5] Tucker, S. G., " Microprogram-Control for System/360 ", IBM Systems Journal, V.6, No.4, 1967, pp.222-241.

[6] Hassitt, A., J. Lageschulte and L.E. Lyon, "Implementation of A High Level Language Machine," CACM, 16, No.4, April 1973, pp.199-212.

[7] Weber, H., "A Microprogrammed Implementation of EULER on IBM S/360 Model 30," CACM, 10 No.9, Sept. 1967, pp.549-558.

[8] Werkheiser, A.H., "Microprogrammed Operating System," 3rd Annu. Workshop on Microprogramming (preprints), ACM October 1970.

[9] Liskov, B.H., "The Design of the Venus Operating System," CACM, 15, No.3, March 1972, pp.144-149.

[10] Agrawala, A.K. and T.G. Rauscher, Foundation of Microprogramming, Academic Press, New York, 1976.

[11] Mallett, P. W. and T. G. Lewis, " Approaches to Design of High Level Languages for Microprogramming ", Proc. of the

7th Annual Workshop on Microprogramming, 1974, pp.66-73.

[12] Knuth, D. E., " Structured programming with GOTO Statements ", Computing Survey, V.5, NO.4, 1974, pp.261-301.

[13] IEEE Trans. Comput. (special issue on microprogramming), V.C-30.

[14] Iversion, K. E. and A. D. Falkoff, "The Design of APL", IBM J. Res. Develop., Sept., 1973.

[15] Lathwell, R. H. and J. E. Mezei, "A Formal Description of APL", IRIA Colloque APL, Sept., 1971, pp.181-215.

[16] Dasgupta, S., "Towards a Microprogramming Language Schema," Proc. 11th Annu. Workshop on Microprogramming, pp.144-153.

[17] IBM manual no. SB21-3039-0 on APL2.

[18] Kelly, R.A., "APLGOL, an Experimental Structured Programming Language," IBM J. Res. & Dev., Jan., 1973, pp.69-73.

Appendix A The Emulator Listing

```
 1 MPG 'EMULATOR'
 2
 3 DECLARATION
 4         'OPCODE' SYNTO 'IR∆OP'
 5         'INDEX' SYNTO 'IR∆R1'
 6         'I1' SYNTO 'IR∆R2'
 7         'I2' SYNTO 'IR∆B2'
 8         'XRN' SYNTO 'INDC,7'
 9         'ARN' SYNTO 'INDC,6'
10         'ARZ' SYNTO 'INDC,5'
11         'AOV' SYNTO 'INDC,4'
12         'STKM' SYNTO 'INDC,3'
13         'STKF' SYNTO 'INDC,2'
14         'IO' SYNTO 'INDC,1'
15         'ENAB' SYNTO 'INDC,0'
16
17           MACRO 'LOAD A,B'
18           B TO MAR
19           1 TO MOP
20           MDR TO A
21           ENDMACRO
22
23           MACRO 'STOR A,B'
24           B TO MAR
25           A TO MDR
26           2 TO MOP
27           ENDMACRO
28
29           MACRO 'LOAD∆CS A,B'
30           B TO MMAR
31           1 TO MOP
32           MMDR TO A
33           ENDMACRO
34
35           MACRO 'STOR∆CS A,B'
36           B TO MMAR
37           A TO MMDR
38           2 TO MOP
39           ENDMACRO
40
41           MACRO 'LOAD∆MXR N,B'
42           N TO MXRAR
43           1 TO MXROP
44           MXRDR TO B
45           ENDMACRO
46
47           MACRO 'STOR∆MXR N,B'
48           N TO MXRAR
49           B TO MXRDR
50           2 TO MXROP
51           ENDMACRO
52
```

(cont'd)

```
 53          MACRO 'LOADΔXR N,B'
 54          N TO XRAR
 55          1 TO MXROP
 56          XRDR TO B
 57          ENDMACRO
 58
 59          MACRO 'STORΔXR N,B'
 60          N TO XRAR
 61          B TO XRDR
 62          2 TO XROP
 63          ENDMACRO
 64
 65          MACRO 'LOADΔFR N,B'
 66          N TO FRAR
 67          1 TO FROP
 68          FRDR TO B
 69          ENDMACRO
 70
 71          MACRO 'STORΔFR N,B'
 72          N TO FRAR
 73          B TO FRDR
 74          2 TO FROP
 75          ENDMACRO
 76
 77          MACRO 'INFET'
 78          PC TO MAR
 79          1 TO MOP
 80          MDR TO IR
 81          ENDMACRO
 82
 83          MACRO 'GENADD'
 84          IRΔB2 TO XRDR
 85          1 TO XROP
 86          80 TO XOP
 87          XRDR TO MAR
 88          ENDMACRO
 89
 90          MACRO 'FDIV INDEX,MDR'
 91          INDEX TO FRDR
 92          1 TO FROP
 93          (HEX '7D') TO AOP
 94          RESULT FRDR
 95          ENDMACRO
 96
 97          MACRO 'FADD INDEX,MDR'
 98          INDEX TO FRDR
 99          1 TO FROP
100          (HEX '7A') TO AOP
101          RESULT FRDR
102          ENDMACRO
103
```

(Cont'd)

```
104         MACRO 'FSUB INDEX,MDR'
105         INDEX TO FRAR
106         1 TO FROP
107         (HEX '7B') TO AOP
108         RESULT FRDR
109         ENDMACRO
110
111         MACRO 'FMUL INDEX,MDR'
112         INDEX TO FRDR
113         1 TO FROP
114         (HEX '7C') TO AOP
115         RESULT FRDR
116         ENDMACRO
117         MACRO 'BADD XRDR,MDR'
118         90 TO XOP
119         RESULT 'XRDR'  .
120         ENDMACRO
121
122         MACRO 'BSUB XRDR,MDR'
123         91 TO XOP
124         RESULT XRDR
125         ENDMACRO
126
127         MACRO 'BMUL XRDR,MDR'
128         92 TO XOP
129         RESULT XRDR
130         ENDMACRO
131
132         MACRO 'BDIV XRDR,MDR'
133         93 TO XOP
134         RESULT XRDR
135         ENDMACRO
136
137         MACRO 'AND XRDR,MDR'
138         84 TO XOP
139         RESULT XRDR
140         ENDMACRO
141
142         MACRO 'OR XRDR,MDR'
143         86 TO XOP
144         RESULT XRDR
145         ENDMACRO
146
147         MACRO 'NAND XRDR,MDR'
148         94 TO XOP
149         RESULT XRDR
150         ENDMACRO
151
152         MACRO 'NOR XRDR,MDR'
153         95 TO XOP
154         RESULT XRDR
155         ENDMACRO
156
```

(Cont'd)

```
157           MACRO 'ADD XRDR,IRΔD2'
158           80 TO XOP
159           RESULT XRDR
160           ENDMACRO
161
162           MACRO 'DEC XRDR'
163           82 TO XOP
164           RESULT XRDR
165           ENDMACRO
166
167           MACRO 'INC XRDR'
168           83 TO XOP
169           RESULT XRDR
170           ENDMACRO
171
172           MACRO 'SHAL XRDR,N'
173           REPEAT 137 TO XOP Δ M1 'N'
174           →UNTIL 'N' EQ 0
175           RESULT XRDR
176           ENDMACRO
177
178           MACRO 'SHAR XRDR,N'
179           REPEAT 136 TO XOP Δ M1 'N'
180           →UNTIL 'N' EQ 0
181           RESULT XRDR
182           ENDMACRO
183
184           MACRO 'SHLL XRDR,N'
185           REPEAT 142 TO XOP Δ M1 'N'
186           →UNTIL 'N' EQ 0
187           RESULT XRDR
188           ENDMACRO
189
190 ENDEC
191
192 BEGIN
193         PROC 'INIT'
194           BEGIN
195             (HEX '80') TO INDC
196             (HEX 'FF') TO MASK
197           END
198         ENDPROC
199
200         PROC 'START'
201           BEGIN
202             INFET
203             GENADD
204           END
205         ENDPROC
206
207 DCODE: OPCODE TO MPC
208
```

(Cont'd)

```
209         PROC 'STE'
210           DECL
211             'EA' SYNTO 'MAR'
212           ENDECL
213           BEGIN
214             LOADΔFR INDEX,FRDR
215             STOR UFRDR,EA
216             ACT 'IPC'
217           END
218         ENDPROC
219
220         PROC 'ST'
221           DECL
222             'EA' SYNTO 'MAR'
223           ENDECL
224           BEGIN
225             LOADΔXR INDEX,XRDR
226             STOR XRDR,EA
227             ACT 'IPC'
228           END
229         ENDPROC
230
231         PROC 'STDX'
232           DECL
233             'EA' SYNTO 'MAR'
234           ENDECL
235           BEGIN
236             LOADΔXR INDEX,XRDR
237             STOR XRDR,EA
238             ADD1 EA
239             ADD1 INDEX
240             LOADΔXR INDEX,XRDR
241             STOR XRDR,EA
242             ACT 'IPC'
243           END
244         ENDPROC
245
246         PROC 'LD'
247           DECL
248             'EA' SYNTO 'MAR'
249           ENDECL
250           BEGIN
251             LOAD MDR,EA
252             STORΔXR INDEX,MDR
253             ACT 'IPC'
254           END
255         ENDPROC
256
257         PROC 'AE'
258           DECL
259             'EA' SYNTO 'MAR'
260           ENDECL
261           BEGIN
262             LOAD MDR,EA
263             FADD INDEX,MDR
264             STORΔFR INDEX,FRDR
265             ACT 'IPC'
266           END
267         ENDPROC
268
```

(Cont'd)

```
269         PROC 'SE'
270           DECL
271             'EA' SYNTO 'MAR'
272           ENDECL
273           BEGIN
274             LOAD MDR,EA
275             FSUB INDEX,MDR
276             STOR∆FR INDEX,FRDR
277             ACT 'IPC'
278           END
279         ENDPROC
280
281         PROC 'ME'
282           DECL
283             'EA' SYNTO 'MAR'
284           ENDECL
285           BEGIN
286             LOAD MDR,EA
287             FMUL INDEX,MDR
288             STOR∆FR INDEX,FRDR
289             ACT 'IPC'
290           END
291         ENDPROC
292
293         PROC 'DE'
294           DECL
295             'EA' SYNTO 'MAR'
296           ENDECL
297           BEGIN
298             LOAD MDR,EA
299             FDIV INDEX,MDR
300             STOR∆FR INDEX,FRDR
301             ACT 'IPC'
302           END
303         ENDPROC
304
305         PROC 'M'
306           DECL
307             'EA' SYNTO 'MAR'
308           ENDECL
309           BEGIN
310             LOAD MDR,EA
311             BMUL INDEX,MDR
312             STOR∆XR INDEX,XRDR
313             ACT 'IPC'
314           END
315         ENDPROC
316
317         PROC 'SI'
318           DECL
319             'IMME' SYNTO 'MAR'
320           ENDECL
321           BEGIN
322             BSUB INDEX,IMME
323             STOR∆XR INDEX,XRDR
324             ACT 'IPC'
325           END
326         ENDPROC
327
328         PROC 'DI'
329           DECL
330             'IMME' SYNTO 'MAR'
331           ENDECL
```

(Cont'd)

```
332           BEGIN
333             BDIV INDEX,IMME
334             STORΔXR INDEX,XRDR
335             ACT 'IPC'
336           END
337         ENDPROC
338
339         PROC 'NA'
340           DECL
341             'EA' SYNTO 'MAR'
342           ENDECL
343           BEGIN
344             LOAD MDR,EA
345             NAND INDEX,MDR
346             ACT 'IPC'
347           END
348         ENDPROC
349
350         PROC 'NOI'
351           DECL
352             'IMME' SYNTO 'MAR'
353           ENDECL
354           BEGIN
355             NOR INDEX,IMME
356             ACT 'IPC'
357           END
358         ENDPROC
359
360         PROC 'SRL'
361           BEGIN
362             SHAR INDEX,1
363             ACT 'IPC'
364           END
365         ENDPROC
366
367         PROC 'ROL'
368           BEGIN
369             SHAL INDEX,1
370             ACT 'IPC'
371           END
372         ENDPROC
373
374         PROC 'CAL'
375           DECL
376             'EA' SYNTO 'MAR'
377           ENDECL
378           BEGIN
379             PSW TO STACK
380             EA TO PC
381             ACT 'INTRUP'
382           END
383         ENDPROC
384
385         PROC 'RETN'
386           BEGIN
387             STACK TO PSW
388             ACT 'INTRUP'
389           END
390         ENDPROC
391
```

(Cont'd)

```
392         PROC 'BCT'
393           BEGIN
394             LOADΔXR INDEX,XRDR
395             DEC XRDR
396             STORΔXR INDEX,XRDR
397             IF 'IF1 ARZ'
398             THEN ACT 'IPC' WITH RESET
399             ENDIF
400             MAR TO PC
401             ACT 'INTRUP'
402           END
403         ENDPROC
404
405         PROC 'B'
406           DECL
407             'EA' SYNTO 'MAR'
408           ENDECL
409           BEGIN
410             EA TO PC
411           END
412         ENDPROC
413
414         PROC 'BSTKF'
415           BEGIN
416             IF 'IF1 INDC,2'
417               THEN GOTO 'BRANCH'
418             ENDIF
419             IF 'IF0 IR,11'
420               THEN ACT 'IPC'
421             ENDIF
422             IF 'IF0 INDC,2'
423             THEN ACT 'IPC' WITH REV
424             ENDIF
425 BRANCH:    IF 'IF1 IR,10'
426               THEN ACT 'REPL'
427             ENDIF
428             IF 'IF1 INDC,2'
429             THEN ACT 'REPL' WITH RESET
430             ENDIF
431           END
432         ENDPROC
433
434         PROC 'BARN'
435           BEGIN
436             IF 'IF1 ARN'
437               THEN GOTO 'BR'
```

(Cont'd)

```
437                 THEN GOTO 'BR'
438               ENDIF
439               IF 'IF0 IR,11'
440                 THEN ACT 'IPC'
441               ENDIF
442               IF 'IF0 INDC,6'
443                 THEN ACT 'IPC' ∧ REV
444               ENDIF
445       BR:     IF 'IF1 IR,10'
446                 THEN ACT 'REPL'
447               ENDIF
448               IF 'IF1 ARN'
449                 THEN ACT 'REPL' ∧ RESET
450               ENDIF
451             END
452           ENDPROC
453
454           PROC 'HALT'
455             BEGIN
456               PC TO PC
457             END
458           ENDPROC
459
460           PROC 'NOOP'
461             BEGIN
462                 ADD1 PC
463             END
464           ENDPROC
465
466           PROC 'SMASK'
467             BEGIN
468               MASK TO XRDR
469               OR XRDR,IR∆D2
470               XRDR TO MASK
471               ACT 'IPC'
472             END
473           ENDPROC
474
475           PROC 'SINDC'
476             BEGIN
477               INDC TO XRDR
478               OR XRDR,IR∆D2
479               XRDR TO INDC
480               ACT 'IPC'
481             END
482           ENDPROC
483
484           PROC 'STMSK'
485             BEGIN
486               STOR∆XR INDEX,MASK
487               ACT 'IPC'
488             END
489           ENDPROC
490
```

(Cont'd)

```
491         PROC 'LDMSK'
492           BEGIN
493             LOADAXR INDEX,MASK
494             ACT 'IPC'
495           END
496         ENDPROC
497
498         PROC 'STIND'
499           BEGIN
500             STORAXR INDEX,INDC
501             ACT 'IPC'
502           END
503         ENDPROC
504
505         PROC 'IPC'
506           BEGIN
507             ADD1 PC
508           END
509         ENDPROC
510
511         PROC 'INTRUP'
512           BEGIN
513             IF 'IF0 INDC,0'
514               THEN ACT 'START'
515             ENDIF
516             IF 'IF0 INDC,1'
517               THEN GOTO 'STKFL'
518             ENDIF
519             IF 'IF0 MASK,1'
520               THEN GOTO 'STKFL'
521             ENDIF
522             ACT 'HALT'
523           END
524         ENDPROC
525
526         PROC 'REPL'
527           BEGIN
528             MAR TO PC
529             ACT 'INTRUP'
530           END
531         ENDPROC
532
533 STKFL: IF 'IF0 STKF'
534         THEN GOTO 'STKMP'
535         ENDIF
536
537         IF 'IF0 MSTKF'
538         THEN GOTO 'STKMP'
539         ENDIF
540         ACT 'HALT'
541
542 STKMP: IF 'IF0 STKM'
543         THEN GOTO 'AOVF'
544         ENDIF
545
546         IF 'IF0 MSTKM'
547         THEN GOTO 'AOVF'
548         ENDIF
549         ACT 'HALT'
550
```

(Cont'd)

```
551 AOVF:   IF 'IF0 AOV'
552         THEN GOTO 'XRNG'
553         ENDIF
554         IF 'IF0 MAOV'
555         THEN GOTO 'XRNG'
556         ENDIF
557         ACT 'HALT'
558
559 XRNG:   IF 'IF0 XRN'
560         THEN ACT 'START'
561         ENDIF
562         ACT 'HALT'
563
564       ACT 'STE' Δ LOC HEX '70'
565       ACT 'ST' Δ LOC HEX '50'
566       ACT 'STDX' Δ LOC HEX '51'
567       ACT 'LD' Δ LOC HEX '58'
568       ACT 'AE' Δ LOC HEX '7A'
569       ACT 'SE' Δ LOC HEX '7B'
570       ACT 'ME' Δ LOC HEX '7C'
571       ACT 'DE' Δ LOC HEX '7D'
572       ACT 'M' Δ LOC HEX '5C'
573       ACT 'SI' Δ LOC HEX '6B'
574       ACT 'NA' Δ LOC HEX '5E'
575       ACT 'NOI' Δ LOC HEX '6F'
576       ACT 'SRL' Δ LOC HEX '88'
577       ACT 'ROL' Δ LOC HEX '8E'
578       ACT 'CAL' Δ LOC HEX '43'
579       ACT 'RETN' Δ LOC HEX '44'
580       ACT 'BCT' Δ LOC HEX '46'
581       ACT 'B' Δ LOC HEX '45'
582       ACT 'BSTKF' Δ LOC HEX '48'
583       ACT 'BARN' Δ LOC HEX '4C'
584       ACT 'HALT' Δ LOC HEX '8A'
585 END
586 ENDMPG
```

Remarks

. MXR(3) is assumed having a value of 256.

. Some keywords (and symbols) other than previously defined are employed for writing the emulator in VS/APL, these keywords (and symbols) are

(a) RESULT: to indicate the return result register

(b) M1 : perform ⁻1 on a literal constant

(c) Δ : a conjunction operator between two operations in the same source statement

(d) WITH : same meaning/function as 'Δ'

(e) LOC : fix the location of the current statement in a particular location of the control store

(f) → : a VS/APL control transfer symbol

There introduction is solely for VS/APL programming conveniences and poses no distortion to the overall AMPL(C) structure.

Appendix B. The Microcode Listing

ADDRESS	*MICROCODE*	/Comment
		/addresses 0-144 are reserved
0145	000100101000000010000000	/ proc. INIT
0146	000101001000000011111111	
0147	000111100000000000001110	/ proc. START
0148	000010001000000000000001	
0149	000100000000000000100000	
0150	001110100000000000011001	
0151	000001101000000000000001	
0152	000001001000000001010000	
0153	000111100000000000011101	
0154	001001100000000000011100	/ OPCODE TO MPC
0155	001000000000000000011101	/ proc. STE
0156	000000101000000000000001	
0157	000101100000000000001011	
0158	000111100000000000001111	
0159	010000000000000000011110	
0160	000010001000000000000010	
0161	100100000000001100110101	
0162	001000100000000000011101	/ proc. ST
0163	000011001000000000000001	
0164	001110100000000000011101	
0165	000111100000000000001111	
0166	010000000000000000011101	
0167	000010001000000000000010	
0168	100100000000001100110101	
0169	001000100000000000011101	/ proc. STDX
0170	000011001000000000000001	
0171	001110100000000000011101	
0172	000111100000000000001111	
0173	010000000000000000011101	
0174	000010001000000000000010	
0175	000111100000000000001111	
0176	001110110000000000011101	
0177	001000100000000000011101	
0178	000011001000000000000001	
0179	001110100000000000011101	
0180	000111100000000000001111	
0181	010000000000000000011101	
0182	000010001000000000000010	
0183	100100000000001100110101	
0184	000111100000000000001111	/ proc. LD
0185	000010001000000000000001	
0186	010000000000000000100000	
0187	001000100000000000011101	
0188	001110100000000000100000	
0189	000001101000000000000010	

(Cont'd)

```
0190        1001000000000011001101 01
0191        0001111000000000000011 11
0192        0000100010000000000000 01
0193        0100000000000000001000 00
0194        0001011000000000000111 01
0195        0000001010000000000000 01
0196        0100100010000000011110 10
0197        0010000000000000000111 01
0198        0001011000000000000010 11
0199        0000001010000000000000 10
0200        1001000000000011001101 01
0201        0001111000000000000011 11        / proc. SE
0202        0000100010000000000000 01
0203        0100000000000000001000 00
0204        0010000000000000000111 01
0205        0000001010000000000000 01
0206        0100100010000000011110 11
0207        0010000000000000000111 01
0208        0001011000000000000010 11
0209        0000001010000000000000 10
0210        1001000000000011001101 01
0211        0001111000000000000011 11        / proc. ME
0212        0000100010000000000000 01
0213        0100000000000000001000 00
0214        0001011000000000000111 01
0215        0000001010000000000000 01
0216        0100100010000000011111 00
0217        0010000000000000000111 01
0218        0001011000000000000010 11
0219        0000001010000000000000 10
0220        1001000000000011001101 01
0221        0001111000000000000011 11        / proc. DE
0222        0000100010000000000000 01
0223        0100000000000000001000 00
0224        0001011000000000000111 01
0225        0000001010000000000000 01
0226        0100100010000000011111 01
0227        0010000000000000000111 01
0228        0001011000000000000010 11
0229        0000001010000000000000 10
0230        1001000000000011001101 01
0231        0001111000000000000011 11        / proc. M
0232        0000100010000000000000 01
0233        0100000000000000001000 00
0234        0000010010000000010111 00
0235        0010001000000000000111 01
0236        0011101000000000000111 01
0237        0000011010000000000000 10
0238        1001000000000011001101 01
0239        0000010010000000010111 11        / proc. SI
0240        0010001000000000000111 01
0241        0011101000000000000111 01
0242        0000011010000000000000 10
0243        1001000000000011001101 01
0244        0000010010000000010111 01        / proc. DI
0245        0010001000000000000111 01
0246        0011101000000000000111 01
0247        0000011010000000000000 10
0248        1001000000000011001101 01
```

(Cont'd)

```
0249  000111100000000000001111          / proc. NA
0250  000010001000000000000001
0251  000111100000000000001111
0252  000001001000000001011110
0253  100100000000001100110101
0254  000001001000000001011111          / proc. NOI
0255  100100000000001100110101
0256  000001001000000010001000          / proc. SRL
0257  100100000000001100110101
0258  000001001000000010001001          /proc. ROL
0259  100100000000001100110101
0260  000010100000000000100001          / proc. CAL
0261  000111000000000000001111
0262  100100000000001100110110
0263  010000100000000000000101          / proc. RETN
0264  100100000000001100110110
0265  001000100000000000011101          / proc. ECT
0266  000011001000000000000001
0267  001110100000000000011101
0268  000001001000000001010010
0269  001000100000000000011101
0270  001110100000000000011101
0271  000001101000000000000010
0272  101000111010101100110101
0273  000111000000000000001111
0274  100000000000001100110110
0275  000111000000000000001111          / proc. B
0276  101000111001000000000000          / proc. BSTKF
0277  100001000101101100110101
0278  100010111001001100110101
0279  101001000101001100111010
0280  101000111001001100111010
0281  101000111011000000000000          / proc. BARN
0282  100001000101101100110101
0283  100010111011001100110101
0284  101001000101001100111010
0285  101000111011001100111010
0286  000111000000000000001110          /proc. HALT
0287  000111010000000000001110          / proc. NOOP
0288  001110100000000000001010          /  proc. MASK
0289  000001001000000001010110
0290  000101000000000000011101
0291  100000000000001100110101
0292  001110100000000000001001          / proc. SINDC
0293  000001001000000001010110
0294  000100100000000000011101
0295  100000000000001100110101
0296  001000100000000000011101          / proc. STMSK
0297  001110100000000000001010
0298  000001101000000000000010
0299  100000000000001100110101
0300  001000100000000000011101          / proc. LDMSK
0301  000011001000000000000001
0302  000101000000000000011101
0303  100000000000001100110101
0304  001000100000000000011101          / proc. STIND
0305  001110100000000000001001
0306  000001101000000000000010
0307  100000000000001100110101
0308  000111010000000000011110          / proc. IPC
0309  100000111000000010010011          / proc. INTRUP
0310  100000111000100000000000
0311  100000110000100000000000
0312  100000000000001100011111
0313  000111000000000000001111          / proc. REPL
0314  100000000000001100110110
```

(Cont'd)

```
0315    1000001110010000000000000      / Label: STKFL
0316    1000001100010000000000000
0317    1000000000000011000011111
0318    1000001110011000000000000      / Label: STKMP
0319    1000001100011000000000000
0320    1000000000000011000011111
0321    1000001110100000000000000      / Label: AOVF
0322    1000001100100000000000000
0323    1000000000000011000011111
0324    1000001110111000100100011      / Label: XRNG
0325    1000000000000011000011111
0112    1000000000000000100110111      / Jump Table Statements
0080    1000000000000000101000010
0081    1000000000000000101010011
0088    1000000000000000101110000
0122    1000000000000000101111111
0123    1000000000000000110010011
0124    1000000000000000110100111
0125    1000000000000000110111011
0092    1000000000000000111001111
0107    1000000000000000111011111
0094    1000000000000000111110011
0111    1000000000000000111111110
0136    1000000000000001100000001
0142    1000000000000011000000011
0067    1000000000000001100000101
0068    1000000000000011000001000
0070    1000000000000011000010100
0069    1000000000000000111101111
0072    1000000000000011000101011
0076    1000000000000011000110100
0138    1000000000000011000111111
```

Note: The translating programs are not shown here. However, they can be obtained from the author* on request.

* : K.W.WU, Computing Studies Unit, Hong Kong Baptist College, 224 Waterloo Road, Kowloon, Hong Kong.